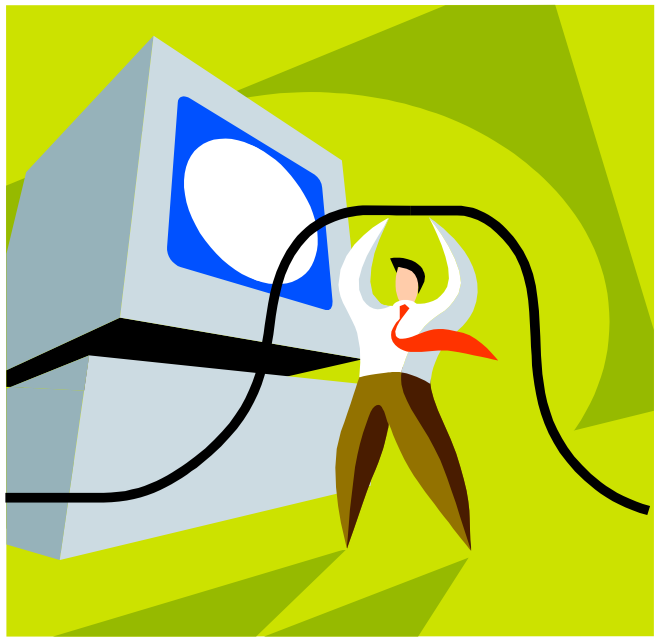
COS 217: Introduction to Programming Systems

Jennifer Rexford

Goals for Today's Class



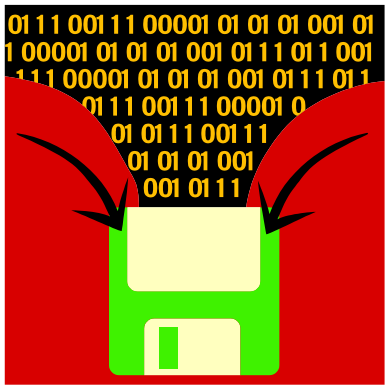
- Course overview
  - Introductions
  - Course goals
  - Resources
  - Grading
  - Policies
- Getting started with C
  - C programming language overview

Introductions

- Lectures
  - Jennifer Rexford (Professor)
    - jrex@cs.princeton.edu
- Preceptors
  - Christopher Moretti (Lead Preceptor)
    - cmoretti@cs.princeton.edu
  - Sibren Isaacman
    - isaacman@princeton.edu

Course Goal 1: "Programming in the Large"

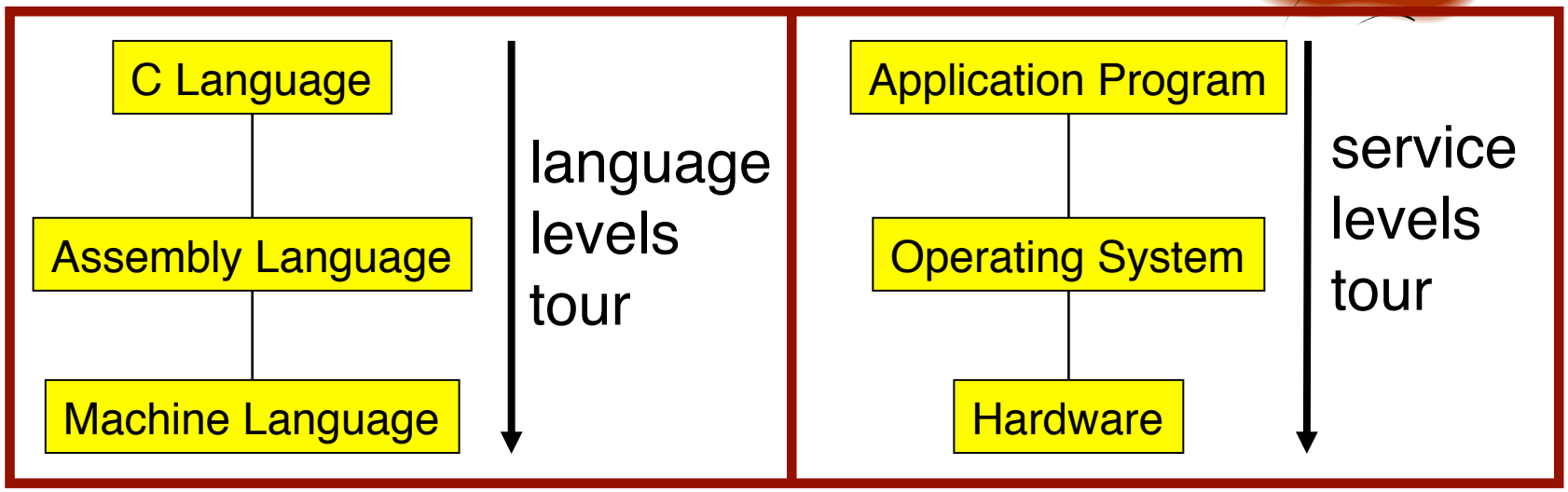
- Help you learn how to write large computer programs



- Specifically:
  - Write modular code
  - Write portable code
  - Test and debug your code
  - Improve your code's performance (and when to do so)
  - Use tools to support those activities

Course Goal 2: "Under the Hood"

- Help you learn what happens "under the hood" of computer systems
- Two downward tours



- Goal 2 supports Goal 1
  - Reveals many examples of effective abstractions

Course Goals: Why C Instead of Java?

- A: C supports Goal 1 better
  - C is a lower-level language
    - C provides more opportunities to create abstractions
  - C has some flaws
    - C's flaws motivate discussions of software engineering principles
- A: C supports Goal 2 better
  - C facilitates language levels tour
    - C is closely related to assembly language
  - C facilitates service levels tour
    - Linux is written in C

Course Goals: Why Linux?

- A: Linux is good for education and research
  - Linux is open-source and well-specified
- A: Linux is good for programming
  - Linux is a variant of Unix
  - Unix has GNU, a rich open-source programming environment

Course Goals: Summary

- Help you to become a...

Power Programmer!!!

Resources: Lectures and Precepts

- Lectures
  - Describe concepts at a high level
  - Slides available online at course Web site
- Precepts
  - Support lectures by describing concepts at a lower level
  - Support your work on assignments
- Note: Precepts begin on Monday (i.e., today)
  - P01: MW 1:30-2:20pm, in CS 102
  - P02: TTh 1:30-2:20pm, in CS 102
  - P03: TTh 7:30-8:20pm, in CS 102

Resources: Website and Piazzza

- Website
  - Access from http://www.cs.princeton.edu
    - Academics → Course Schedule → COS 217
- Discussion forum
  - Piazzza: http://www.piazzza.com
  - "Join or create your class now"
    - School: Princeton University
    - Class: COS 217
    - Fill in your Princeton e-mail address
  - Click "get started" link in your email to activate
    - Please use your real name when signing up

Resources: Books

- Required book
  - *C Programming: A Modern Approach (Second Edition),* King, 2008.
    - Covers the C programming language and standard libraries
- Highly recommended books
  - *The Practice of Programming,* Kernighan and Pike, 1999.
    - Covers "programming in the large" (required for COS 333)
  - *Computer Systems: A Programmer's Perspective (Second Edition)*, Bryant and O'Hallaron, 2010.
    - Covers "under the hood," key sections are on e-reserve
    - First edition is sufficient
  - *Programming with GNU Software*, Loukides and Oram, 1997.
    - Covers tools
- *All books are on reserve in Engineering Library*

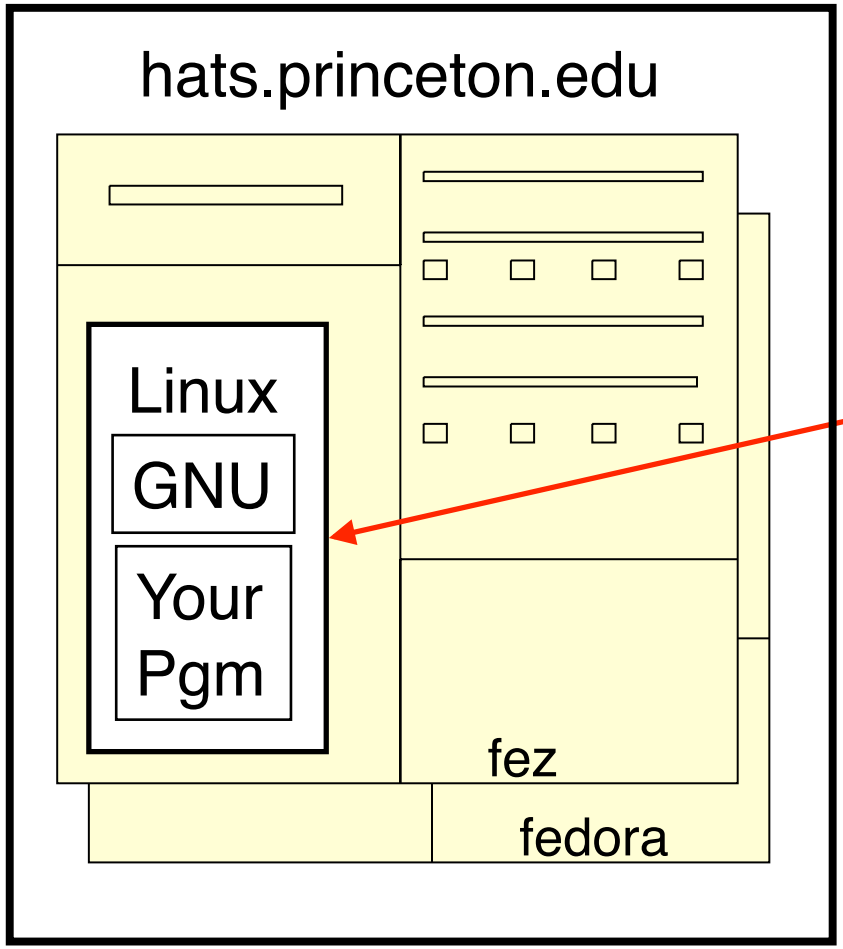
Resources: Manuals

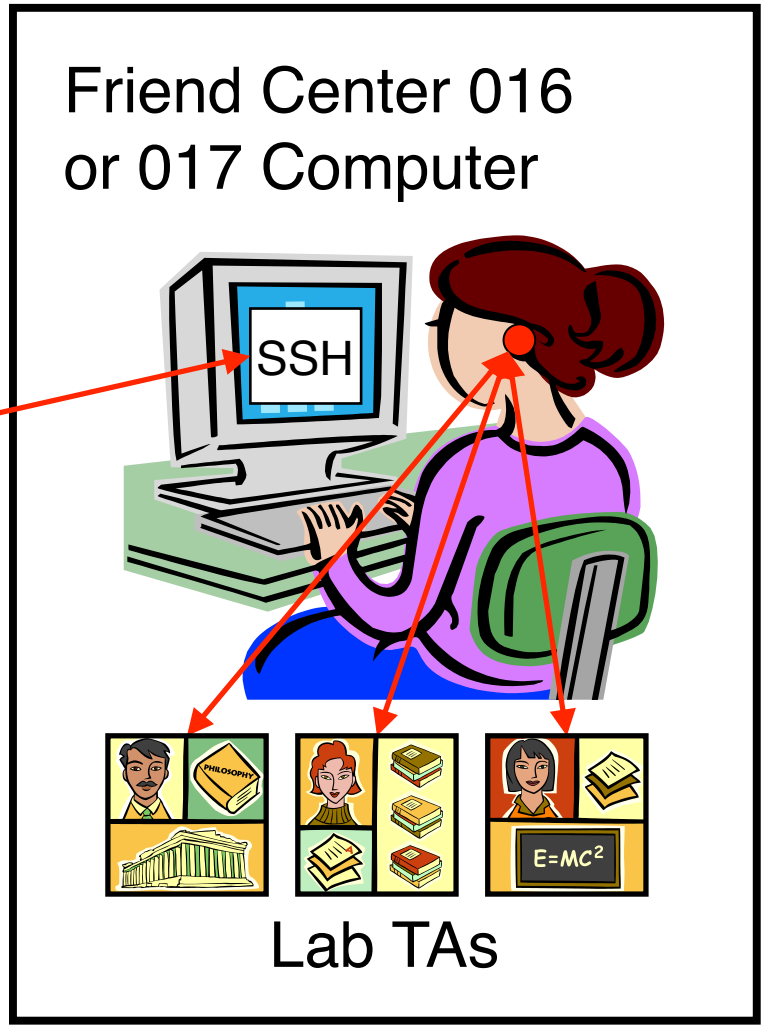
- Manuals (for reference only, available online)
  - *IA32 Intel Architecture Software Developer's Manual, Volumes 1-3*
  - *Tool Interface Standard & Executable and Linking Format*
  - *Using as, the GNU Assembler*

- See also
  - Linux **`man`** command
    - **`man`** is short for “manual”
    - For more help, type **`man man`**

Resources: Programming Environment

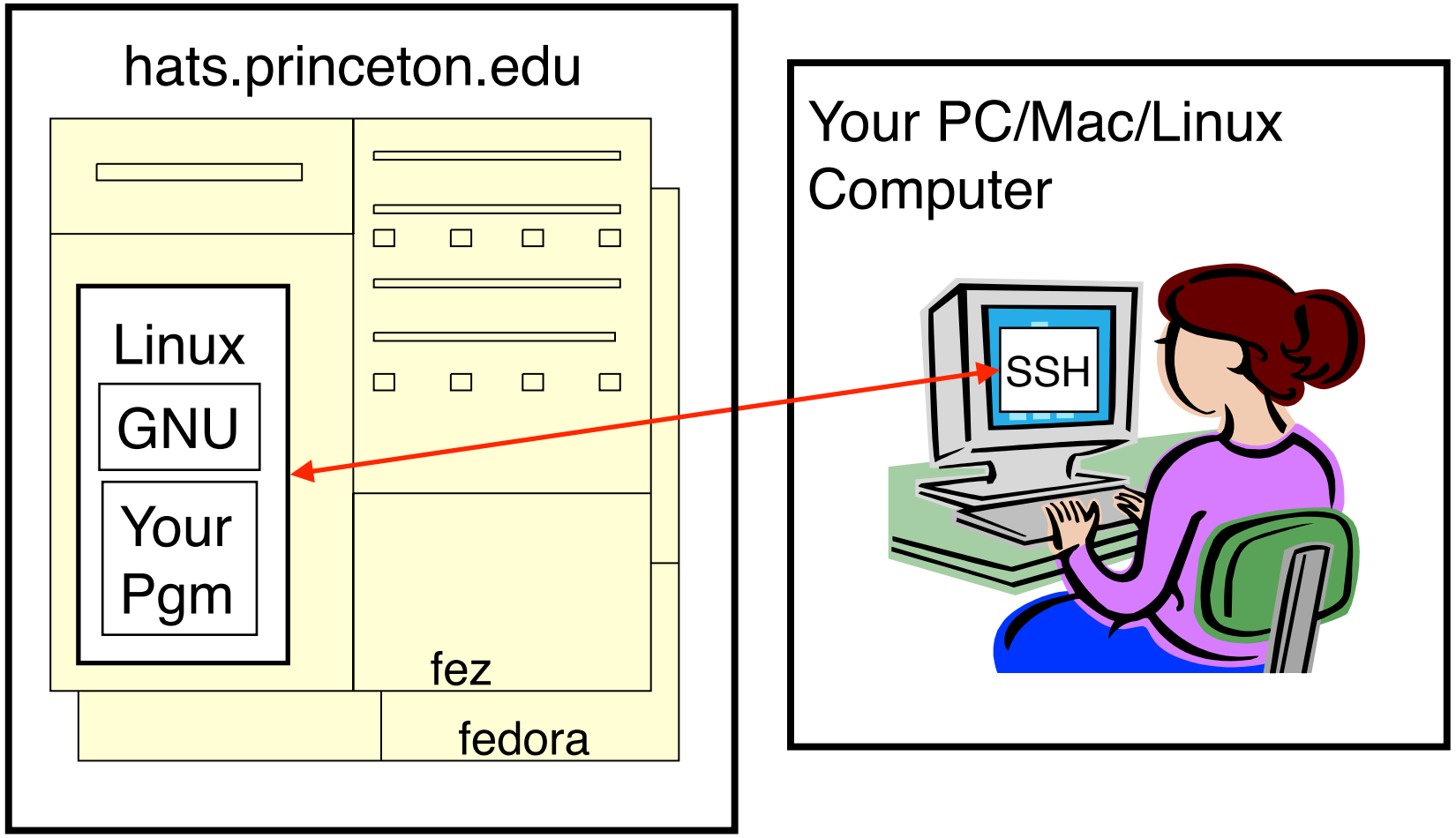
- Option 1





Resources: Programming Environment

- Option 2

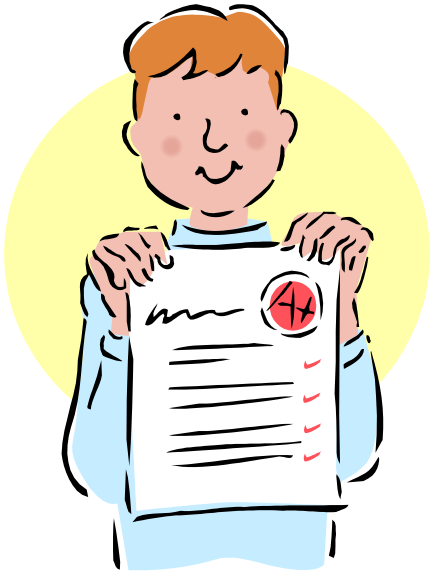


Resources: Programming Environment

- Other options
  - Use your own PC/Mac/Linux computer; run GNU tools locally; run your programs locally
  - Use your own PC/Mac/Linux computer; run a non-GNU development environment locally; run programs locally
- Notes
  - Other options cannot be used for some assignments (esp. timing studies)
  - Instructors cannot promise support of other options
  - Strong recommendation: Use Option 1 or 2 for **all** assignments
  - First precept provides setup instructions

Grading

- Seven programming assignments (50%)
  - Working code
  - Clean, readable, maintainable code
  - On time (penalties for late submission)
  - Final assignment counts double (12.5%)
- Exams (40%)
  - Midterm (15%)
  - Final (25%)
- Class participation (10%)
  - Lecture and precept attendance is ***mandatory***



Programming Assignments

- Programming assignments
  1. A “de-comment” program
  2. A string module
  3. A symbol table module
  4. IA-32 assembly language programs
  5. A buffer overrun attack
  6. A heap manager module
  7. A Unix shell
- See course “Schedule” web page for due dates/times
- Advice: Start early to allow time for debugging (especially in the background while you are doing other things!)…

Why Debugging is Necessary…

Policies

Study the course "Policies" web page!!!

- Especially the assignment collaboration policies
  - Violation involves **trial by Committee on Discipline**
  - Typical penalty is **suspension from University** for 1 academic year
- Some highlights:
  - Don't view anyone else's work during, before, or after the assignment time period
  - Don't allow anyone to view your work during, before, or after the assignment time period
  - In your assignment "readme" file, acknowledge all resources used
- Ask your preceptor for clarifications if necessary

Course Schedule

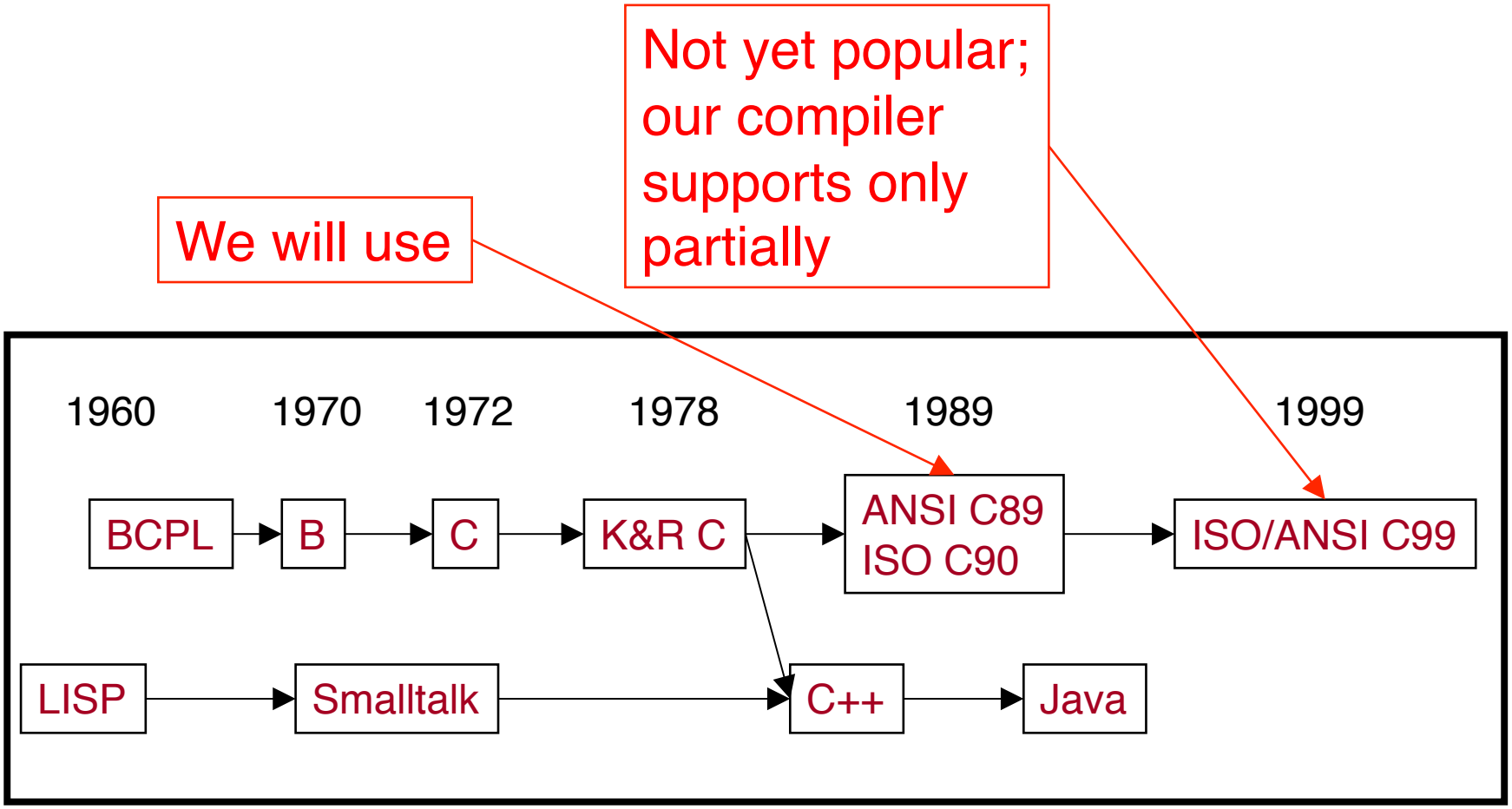
- Very generally…

Weeks	Lectures	Precepts
1-2	Intro to C (conceptual)	Intro to Linux/GNU Intro to C (mechanical)
3-6	"Pgmming in the Large"	Advanced C
6	Midterm Exam	
7	Recess	
8-13	"Under the Hood"	Assembly Language Pgmming Assignments
	Reading Period	
	Final Exam	

- See course "Schedule" web page for details

Any questions before we start?

C vs. Java: History



C vs. Java: Design Goals

- Java design goals
  - Support object-oriented programming
  - Allow same program runs on multiple operating systems
  - Support using computer networks
  - Execute code from remote sources securely
  - Adopt the good parts of other languages
- Implications for Java
  - Good for application-level programming
  - High-level (insulates from assembly language, hardware)
  - Portability over efficiency
  - Security over efficiency
  - Security over flexibility

C vs. Java: Design Goals

- C design goals
  - Support structured programming
  - Support development of the Unix OS and Unix tools
    - As Unix became popular, so did C
- Implications for C
  - Good for system-level programming
  - Low-level
  - Efficiency over portability
  - Efficiency over security
  - Flexibility over security

C vs. Java: Design Goals

- Differences in design goals explain many differences between the languages
- C's design goal explains many of its eccentricities
  - We'll see examples throughout the course

C vs. Java: Overview

- Dennis Ritchie on the nature of C:

  - "C has always been a language that never attempts to tie a programmer down."
  - "C has always appealed to systems programmers who like the terse, concise manner in which powerful expressions can be coded."
  - "C allowed programmers to (while sacrificing portability) have direct access to many machine-level features that would otherwise require the use of assembly language."
  - "C is quirky, flawed, and an enormous success. While accidents of history surely helped, it evidently satisfied a need for a system implementation language efficient enough to displace assembly language, yet sufficiently abstract and fluent to describe algorithms and interactions in a wide variety of environments."

C vs. Java: Overview (cont.)

- Bad things you **can** do in C that you **can't** do in Java
  - Shoot yourself in the foot (safety)
  - Shoot others in the foot (security)
  - Ignore wounds (error handling)
- Dangerous things you **must** do in C that you **don't** in Java
  - Explicitly manage memory via `malloc()` and `free()`
- Good things you **can** do in C, but (more or less) **must** do in Java
  - Program using the object-oriented style
- Good things you **can't** do in C but **can** do in Java
  - Write completely portable code

C vs. Java: Details

- Remaining slides provide some details
  - Suggestion: Use for future reference

- Slides covered briefly now, as time allows…

C vs. Java: Details (cont.)

	Java	C

Overall Program Structure	**Hello.java:**
	`public class Hello {`
	`  public static void`
	`  main(String[] args) {`
	`  System.out.println(`
	`  "Hello, world");`
	`  }`
	`}`
		hello.c:
		`#include <stdio.h>`
		`int main(void) {`
		`  printf("Hello, world\n");`
		`  return 0;`
		`}`

Building	`% javac Hello.java`
	`% ls`
	`Hello.class`
	`Hello.java`
	`%`
		`% gcc217 hello.c`
		`% ls`
		`a.out`
		`hello.c`
		`%`

Running	`% java Hello`
	`Hello, world`
	`%`
		`% a.out`
		`Hello, world`
		`%`

C vs. Java: Details (cont.)

	Java	C
Character type	`char // 16-bit unicode`	`char /* 8 bits */`

Integral types	`byte // 8 bits`
	`short // 16 bits`
	`int // 32 bits`
	`long // 64 bits`
		`(unsigned) char`
		`(unsigned) short`
		`(unsigned) int`
		`(unsigned) long`

Floating point types	`float // 32 bits`
	`double // 64 bits`
		`float`
		`double`
		`long double`

Logical type	`boolean`	`/* no equivalent */` `/* use integral type */`
Generic pointer type	`// no equivalent`	`void*`

Constants	`final int MAX = 1000;`	`#define MAX 1000`
		`const int MAX = 1000;`
		`enum {MAX = 1000};`

C vs. Java: Details (cont.)

	Java	C

Arrays	`int [] a = new int [10];`
	`float [][] b =`
	`  new float [5][20];`
		`int a[10];`
		`float b[5][20];`

Array bound checking	`// run-time check`	`/* no run-time check */`
Pointer type	`// Object reference is an` `// implicit pointer`	`int *p;`

Record type	`class Mine {`
	`  int x;`
	`  float y;`
	`}`
		`struct Mine {`
		`  int x;`
		`  float y;`
		`}`

C vs. Java: Details (cont.)

	Java	C

Strings	`String s1 = "Hello";`
	`String s2 = new String("hello");`
		`char *s1 = "Hello";`
		`char s2[6];`
		`strcpy(s2, "hello");`

String concatenation	`s1 + s2` `s1 += s2`	`#include <string.h>` `strcat(s1, s2);`
Logical ops	`&&, ||, !`	`&&, ||, !`
Relational ops	`=, !=, >, <, >=, <=`	`=, !=, >, <, >=, <=`
Arithmetic ops	`+, -, *, /, %, unary -`	`+, -, *, /, %, unary -`
Bitwise ops	`>>, <<, >>>, &, |, ^`	`>>, <<, &, |, ^`
Assignment ops	`=, *=, /=, +=, -=, <<=, >>=, >>>=, =, ^=, |=, %=`	`=, *=, /=, +=, -=, <<=, >>=, =, ^=, |=, %=`

C vs. Java: Details (cont.)

	Java	C

if stmt	`if (i < 0)`
	`  statement1;`
	`else`
	`  statement2;`
		`if (i < 0)`
		`  statement1;`
		`else`
		`  statement2;`

switch stmt	`switch (i) {`
	`  case 1:`
	`  ...`
	`  break;`
	`  case 2:`
	`  ...`
	`  break;`
	`  default:`
	`  ...`
	`}`
		`switch (i) {`
		`  case 1:`
		`  ...`
		`  break;`
		`  case 2:`
		`  ...`
		`  break;`
		`  default:`
		`  ...`
		`}`

goto stmt	`// no equivalent`	`goto SomeLabel;`

C vs. Java: Details (cont.)

	Java	C

for stmt	`for (int i=0; i<10; i++)`
	`  statement;`
		`int i;`
		`for (i=0; i<10; i++)`
		`  statement;`

while stmt	`while (i < 0)` `  statement;`	`while (i < 0)` `  statement;`

do-while stmt	`do {`
	`  statement;`
	`  …`
	`} while (i < 0)`
		`do {`
		`  statement;`
		`  …`
		`} while (i < 0);`

continue stmt	`continue;`	`continue;`
labeled continue stmt	`continue SomeLabel;`	`/* no equivalent */`
break stmt	`break;`	`break;`
labeled break stmt	`break SomeLabel;`	`/* no equivalent */`

C vs. Java: Details (cont.)

	Java	C
return stmt	`return 5;` `return;`	`return 5;` `return;`

Compound stmt (alias block)	`{`
	`  statement1;`
	`  statement2;`
	`}`
		`{`
		`  statement1;`
		`  statement2;`
		`}`

Exceptions	`throw, try-catch-finally`	`/* no equivalent */`
Comments	`/* comment */` `// another kind`	`/* comment */`

Method / function call	`f(x, y, z);`
	`someObject.f(x, y, z);`
	`SomeClass.f(x, y, z);`
		`f(x, y, z);`

Example C Program

```
#include <stdio.h>
#include <stdlib.h>

const double KMETERS_PER_MILE = 1.609;

int main(void) {
   int miles;
   double kmeters;
   printf("miles: ");
   if (scanf("%d", &miles) != 1) {
      fprintf(stderr, "Error: Expect a number.\n");
      exit(EXIT_FAILURE);
   }
   kmeters = miles * KMETERS_PER_MILE;
   printf("%d miles is %f kilometers.\n",
      miles, kmeters);
   return 0;
}
```

Conclusions

- Getting started with C
  - C was designed for system programming
    - Different design goals from of Java
    - Explains many of C's eccentricities
  - Knowing Java gives you a head start at learning C
    - C is not object-oriented, but many aspects are similar
- Getting started in the course
  - Check out course Web site **soon**
    - Study "Policies" page
    - First assignment
  - Establish a reasonable computing environment **soon**
    - Instructions given in first precept

Getting Started

- Check out course Web site **soon**
  - Study “Policies” page
  - First assignment is available

- Establish a reasonable computing environment **soon**
  - Instructions given in first precept

C Examples

Jennifer Rexford

Goals of this Lecture

- Help you learn about:
  - The fundamentals of C
  - Deterministic finite state automata (DFA)
  - Expectations for programming assignments
- Why?
  - The fundamentals of C provide a foundation for the systematic coverage of C that will follow
  - DFA are useful in many contexts (e.g., Assignment 1)
- How?
  - Through some examples…

Overview of this Lecture

- C programming examples
  - Echo input to output
  - Convert all lowercase letters to uppercase
  - Convert first letter of each word to uppercase
- Glossing over some details related to "pointers"
  - … which will be covered subsequently in the course

Example #1: Echo Input to Output

- Include the Standard Input/Output header file (stdio.h)

```
#include <stdio.h>
```

  - Make declarations of I/O functions available to compiler
  - Allow compiler to check your calls of I/O functions

- Define main() function

```
int main(void) { … }
int main(int argc, char *argv[]) { … }
```

  - Starting point of the program, a standard boilerplate
  - Hand-waving: **`argc`** and **`argv`** are for input arguments

Example #1: Echo Input to Output

- Read a single character

```
c = getchar();
```

  - Read a single character from the “standard input stream” (stdin) and return it

- Write a single character

```
putchar(c);
```

  - Write a single character to the “standard output stream” (stdout)

Putting it All Together

```
#include <stdio.h>

int main(void) {
    int c;

    c = getchar();
    putchar(c);

    return 0;
}
```

Why **int** instead of **char**?

Why return a value?

Read and Write Ten Characters

- Loop to repeat a set of lines (e.g., **`for`** loop)
  - Three expressions: initialization, condition, and increment
  - E.g., start at 0, test for less than 10, and increment per iteration

```
#include <stdio.h>
int main(void) {
  int c, i;

  for (i=0; i<10; i++) {
    c = getchar();
    putchar(c);
  }

  return 0;
}
```

Why not this instead:
for (i = 1; i <= 10; i++)

Read and Write Forever

- Infinite `for` loop
  - Simply leave the expressions blank
  - E.g., `for ( ; ; )`

```
#include <stdio.h>
int main(void) {
  int c;

  for ( ; ; ) {
    c = getchar();
    putchar(c);
  }

  return 0;
}
```

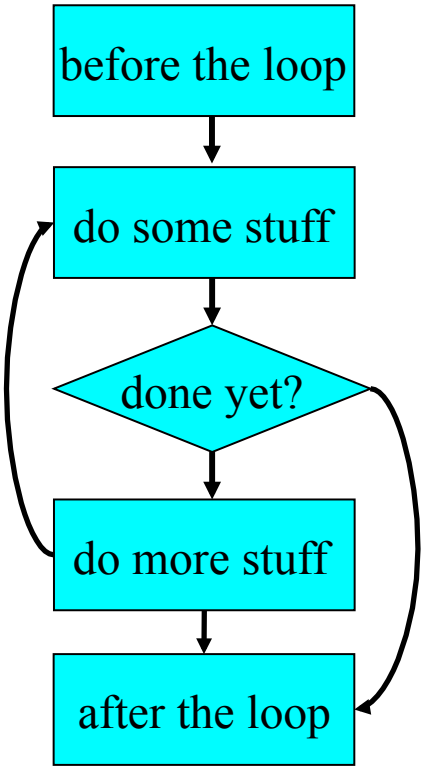
When will this be executed?

How would you terminate this program?

Read and Write Until End-Of-File

- Test for end-of-file
  - **`EOF`** is a global constant, defined in stdio.h
  - The **`break`** statement jumps out of the innermost enclosing loop

```
#include <stdio.h>
int main(void) {
   int c;
   for ( ; ; ) {
      c = getchar();
      if (c == EOF)
         break;
      putchar(c);
   }
   return 0;
}
```



Many Ways to Do the Same Job

```
for (c=getchar(); c!=EOF; c=getchar())
  putchar(c);
```

Which approach is best?

```
while ((c=getchar())!=EOF)
   putchar(c);
```

← Typical idiom in C, but messy side-effect in loop test

```
for (;;)  {
 c = getchar();
 if (c == EOF)
   break;
 putchar(c);
}
```

```
c = getchar();
while (c!=EOF) {
  putchar(c);
  c = getchar();
}
```

Review of Example #1

- Character I/O
  - Including **`stdio.h`**
  - Functions **`getchar()`** and **`putchar()`**
  - Representation of a character as an integer
  - Predefined constant **`EOF`**
- Program control flow
  - The **`for`** and **`while`** statements
  - The **`break`** statement
  - The **`return`** statement
- Operators
  - Assignment operator: **`=`**
  - Increment operator: **`++`**
  - Relational operator to compare for equality: **`==`**
  - Relational operator to compare for inequality: **`!=`**

Example #2: Convert Uppercase

- Problem: Write a program to convert a file to all uppercase
  - Leave non-alphabetic characters alone

- Program design:

```
repeat
  Read a character
  If unsuccessful, break out of loop
  If the character is lower-case, convert to upper-case
  Write the character
```

ASCII

American Standard Code for Information Interchange

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
16	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
32	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Lower case: 97-122 and upper case: 65-90

E.g., ‘a’ is 97 and ‘A’ is 65 (i.e., 32 apart)

Implementation in C

```
#include <stdio.h>
int main(void) {
    int c;
    for ( ; ; ) {
        c = getchar();
        if (c == EOF) break;
        if ((c >= 97) && (c < 123))
            c -= 32;
        putchar(c);
    }
    return 0;
}
```

That's a B-minus

- A good program is:
  - Clean
  - Readable
  - Maintainable
- It's not enough that your program works!
  - We take this seriously in COS 217

Avoid Mysterious Numbers

```
#include <stdio.h>
int main(void) {
    int c;
    for ( ; ; ) {
        c = getchar();
        if (c == EOF) break;
        if ((c >= 97) && (c < 123))
            c -= 32;
        putchar(c);
    }
    return 0;
}
```

Ugly;
ASCII only

Improvement: Character Constants

```
#include <stdio.h>
int main(void) {
    int c;
    for ( ; ; ) {
        c = getchar();
        if (c == EOF) break;
        if ((c >= 'a') && (c <= 'z'))
            c += 'A' - 'a';
        putchar(c);
    }
    return 0;
}
```

Better; but assumes that alphabetic character codes are contiguous

Improvement: Existing Functions

Standard C Library Functions ctype(3C)

NAME

ctype, isdigit, isxdigit, islower, isupper, isalpha, isalnum, isspace, iscntrl, ispunct, isprint, isgraph, isascii - character handling

SYNOPSIS

```
#include <ctype.h>
int isalpha(int c);
int isupper(int c);
int islower(int c);
int isdigit(int c);
int isalnum(int c);
int isspace(int c);
int ispunct(int c);
int isprint(int c);
int isgraph(int c);
int iscntrl(int c);
int toupper(int c);
int tolower(int c);
```

DESCRIPTION

These macros classify character-coded integer values. Each is a predicate returning non-zero for true, 0 for false...

The toupper() function has as a domain a type int, the value of which is representable as an unsigned char or the value of EOF.... If the argument of toupper() represents a lower-case letter ... the result is the corresponding upper-case letter. All other arguments in the domain are returned unchanged.

Using the ctype Functions

```
#include <stdio.h>
#include <ctype.h>
int main(void) {
    int c;
    for ( ; ; ) {
        c = getchar();
        if (c == EOF) break;
        if (islower(c))
            c = toupper(c);
        putchar(c);
    }
    return 0;
}
```

Returns non-zero (true) iff c is a lowercase character

Building and Running

```
% ls
upper.c
% gcc217 upper.c -o upper
% ls
upper upper.c
% upper
We'll be on time today!
WE'LL BE ON TIME TODAY!
^D
%
```

Run the Code on Itself

```
% upper < upper.c
#INCLUDE <STDIO.H>
#INCLUDE <CTYPE.H>
INT MAIN(VOID) {
    INT C;
    FOR ( ; ; ) {
        C = GETCHAR();
        IF (C == EOF) BREAK;
        IF (ISLOWER(C))
            C = TOUPPER(C);
        PUTCHAR(C);
    }
    RETURN 0;
}
```

Output Redirection

```
% upper < upper.c > junk.c
% gcc217 junk.c -o junk
test.c:1:2: invalid preprocessing directive #INCLUDE
test.c:2:2: invalid preprocessing directive #INCLUDE
test.c:3: syntax error before "MAIN"
etc...
```

Review of Example #2

- Representing characters
  - ASCII character set
  - Character constants (e.g., ‘A’ or ‘a’)
- Manipulating characters
  - Arithmetic on characters
  - Functions like **`islower()`** and **`toupper()`**
- Compiling and running C code
  - Compile to generate executable file
  - Invoke executable to run program
  - Can redirect stdin and/or stdout

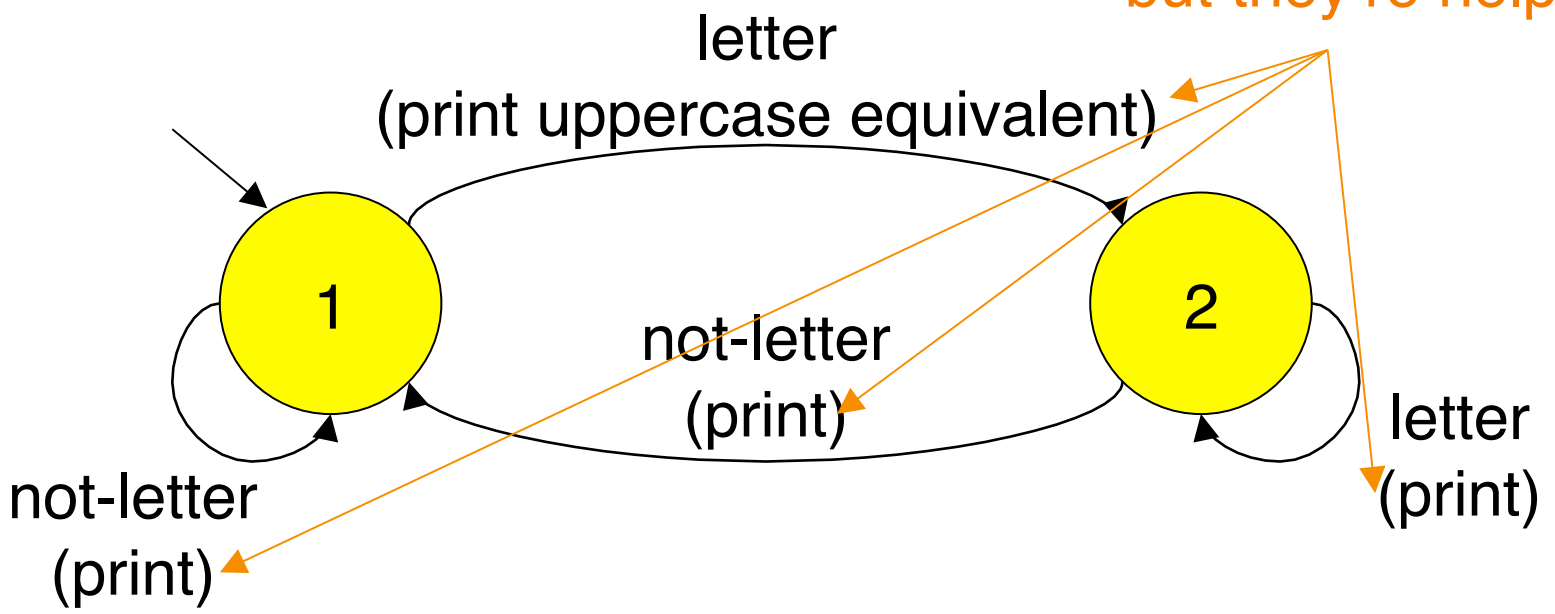
Example #3: Capitalize First Letter

- Capitalize the first letter of each word
  - "cos 217 rocks" → "Cos 217 Rocks"
- Sequence through the string, one letter at a time
  - Print either the character, or the uppercase version
- Challenge: need to remember where you are
  - Capitalize "c" in "cos", but not "o" in "cos" or "c" in "rocks"
- Solution: keep some extra information around
  - Whether you've encountered the first letter in the word

Deterministic Finite Automaton

Deterministic Finite Automaton (DFA)

Actions are not part of DFA formalism; but they're helpful

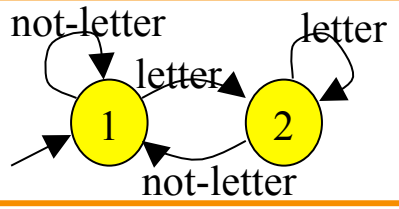


- States
- Transitions labeled by characters (or categories)
- Optionally, transitions labeled by actions

Implementation Skeleton

```
#include <stdio.h>
#include <ctype.h>
int main (void) {
    int c;
    for ( ; ; ) {
        c = getchar();
        if (c == EOF) break;
        <process one character>
    }
    return 0;
}
```

Implementation



```
<process one character> =
switch (state) {
    case 1:
        <state 1 action>
        break;
    case 2:
        <state 2 action>
        break;
    default:
        <this should never happen>
}
```

```
if (isalpha(c)) {
    putchar(toupper(c));
    state = 2;
}
else putchar(c);
```

```
if (!isalpha(c))
    state = 1;
putchar(c);
```

Complete Implementation

```
#include <stdio.h>
#include <ctype.h>
int main(void) {
   int c; int state=1;
   for ( ; ; ) {
      c = getchar();
      if (c == EOF) break;
      switch (state) {
         case 1:
            if (isalpha(c)) {
               putchar(toupper(c));
               state = 2;
            } else putchar(c);
            break;
         case 2:
            if (!isalpha(c)) state = 1;
            putchar(c);
            break;
      }
   }
   return 0;
}
```

Running Code on Itself

```
% gcc217 upper1.c -o upper1
% upper1 < upper1.c
#Include <Stdio.H>
#Include <Ctype.H>
Int Main(Void) {
   Int C; Int State=1;
   For ( ; ; ) {
      C = Getchar();
      If (C == EOF) Break;
      Switch (State) {
         Case 1:
            If (Isalpha(C)) {
               Putchar(Toupper(C));
               State = 2;
            } Else Putchar(C);
            Break;
         Case 2:
            If (!Isalpha(C)) State = 1;
            Putchar(C);
            Break;
      }
   }
   Return 0;
}
```

OK, That's a B

- Works correctly, but
  - Mysterious integer constants ("magic numbers")
- What now?
  - States should have names, not just 1, 2

Improvement: Names for States

- Define your own named constants

```
enum Statetype {NORMAL,INWORD};
```

- Define an enumeration type

```
enum Statetype state;
```

- Define a variable of that type

Improvement: Names for States

```
#include <stdio.h>
#include <ctype.h>
enum Statetype {NORMAL,INWORD};
int main(void) {
   int c; enum Statetype state = NORMAL;
   for ( ; ; ) {
      c = getchar();
      if (c == EOF) break;
      switch (state) {
         case NORMAL:
            if (isalpha(c)) {
               putchar(toupper(c));
               state = INWORD;
            } else putchar(c);
            break;
         case INWORD:
            if (!isalpha(c)) state = NORMAL;
            putchar(c);
            break;
      }
   }
   return 0;
}
```

OK, That's a B+

- Works correctly, but
  - No modularity
- What now?
  - Should handle each state in a separate function

Improvement: Modularity

```
#include <stdio.h>
#include <ctype.h>
enum Statetype {NORMAL,INWORD};
enum Statetype handleNormalState(int c) {...}
enum Statetype handleInwordState(int c) {...}
int main(void) {
    int c;
    enum Statetype state = NORMAL;
    for ( ; ; ) {
        c = getchar();
        if (c == EOF) break;
        switch (state) {
            case NORMAL:
                state = handleNormalState(c);
                break;
            case INWORD:
                state = handleInwordState(c);
                break;
        }
    }
    return 0;
}
```

Improvement: Modularity

```
enum Statetype handleNormalState(int c) {
    enum Statetype state;
    if (isalpha(c)) {
        putchar(toupper(c));
        state = INWORD;
    }
    else {
        putchar(c);
        state = NORMAL;
    }
    return state;
}
```

Improvement: Modularity

```
enum Statetype handleInwordState(int c) {
    enum Statetype state;
    putchar(c);
    if (!isalpha(c))
        state = NORMAL;
    else
        state = INWORD;
    return state;
}
```

OK, That's an A-

- Works correctly, but
  - No comments
- What now?
  - Should add (at least) function-level comments

Function Comments

- A function's comment should:
  - Describe **what the function does**
    - Describe input to the function
      - Parameters, input streams
    - Describe output from the function
      - Return value, output streams, (call-by-reference parameters)
  - **Not** describe **how the function works**

Function Comment Examples

- **Bad** main() function comment

```
Read a character from stdin. Depending upon the
 current DFA state, pass the character to an
 appropriate state-handling function. The value
 returned by the state-handling function is the
 next DFA state. Repeat until end-of-file.
```

  - Describes **how the function works**

- **Good** main() function comment

```
Read text from stdin. Convert the first
 character of each "word" to uppercase, where a
 word is a sequence of letters. Write the result
 to stdout. Return 0.
```

  - Describes **what the function does** from caller's point of view

An “A” Effort

```
#include <stdio.h>
#include <ctype.h>

enum Statetype {NORMAL, INWORD};

/*-------------------------------------------------------------*/
/* handleNormalState: Implement the NORMAL state of the DFA.   */
/* c is the current DFA character.  Return the next state.     */
/*-------------------------------------------------------------*/
enum Statetype handleNormalState(int c) {
   enum Statetype state;
   if (isalpha(c)) {
      putchar(toupper(c));
      state = INWORD;
   }
   else {
      putchar(c);
      state = NORMAL;
   }
   return state;
}
```

An “A” Effort

```
/*-------------------------------------------------------------*/
/* handleInwordState: Implement the INWORD state of the DFA.   */
/* c is the current DFA character.  Return the next state.     */
/*-------------------------------------------------------------*/
enum Statetype handleInwordState(int c) {
   enum Statetype state;
   putchar(c);
   if (!isalpha(c))
      state = NORMAL;
   else
      state = INWORD;
   return state;
}
```

An “A” Effort

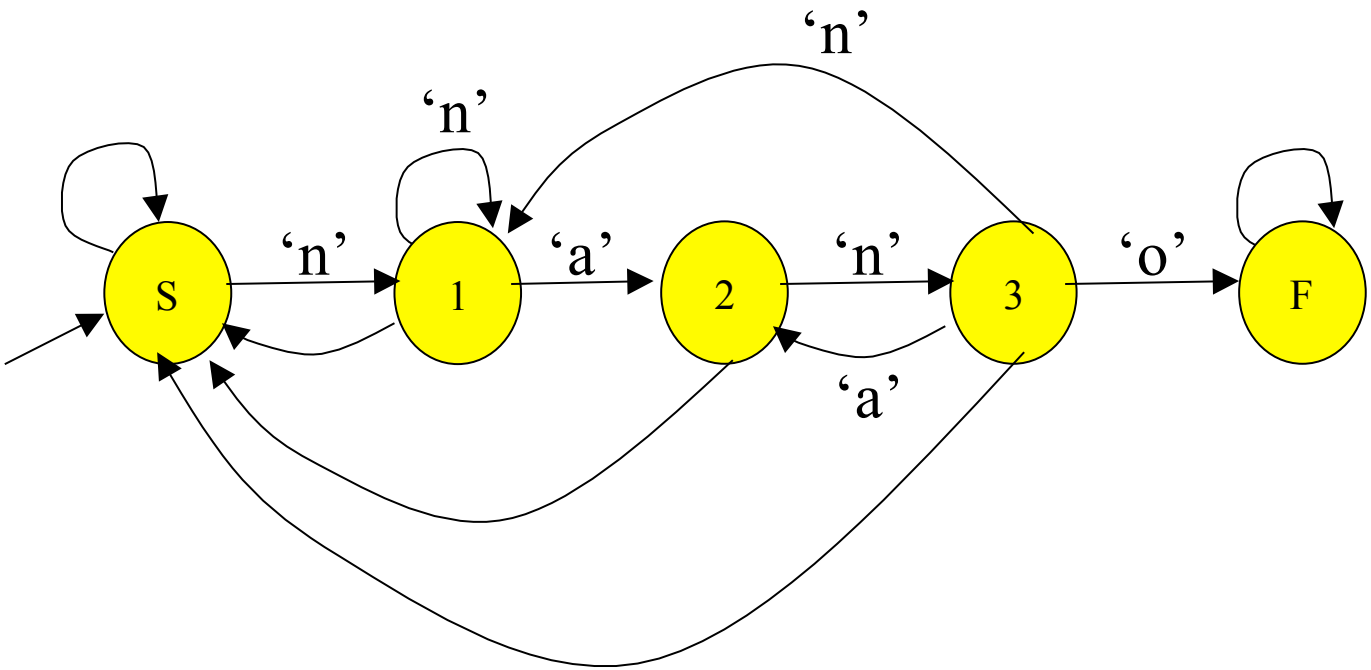
```
/*-------------------------------------------------------------*/
/* main: Read text from stdin. Convert the first character    */
/* of each "word" to uppercase, where a word is a sequence of */
/* letters. Write the result to stdout. Return 0.             */
/*-------------------------------------------------------------*/
int main(void) {
   int c;
   enum Statetype state = NORMAL;
   /* Use a DFA approach.  state indicates the state of the DFA. */
   for ( ; ; ) {
      c = getchar();
      if (c == EOF) break;
      switch (state) {
         case NORMAL:
            state = handleNormalState(c);
            break;
         case INWORD:
            state = handleInwordState(c);
            break;
      }
   }
   return 0;
}
```

Review of Example #3

- Deterministic finite state automaton
  - Two or more states
  - Transitions between states
    - Next state is a function of current state and current character
  - Actions can occur during transitions
- Expectations for COS 217 assignments
  - Readable
    - Meaningful names for variables and values
  - Modular
    - Multiple functions, each of which does one well-defined job
  - Function-level comments
    - Should describe what function does
  - See K&P book for style guidelines specification

Another DFA Example

- Does the string have “nano” in it?
  - “banano” → yes
  - “nnnnnnnanofff” → yes
  - “bananonano” → yes
  - “bananananashanana” → no



Yet Another DFA Example

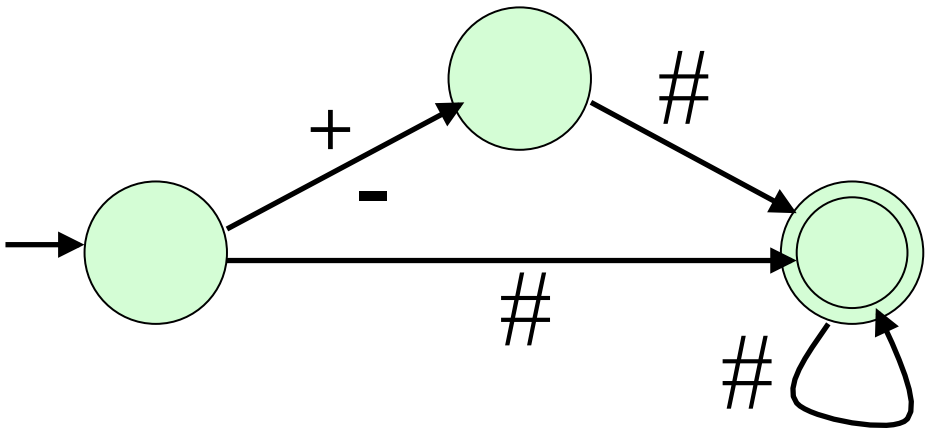
Question #4 from fall 2005 midterm

Identify whether or not a string is a floating-point number

- Valid numbers
  - "-34"
  - "78.1"
  - "+298.3"
  - "-34.7e-1"
  - "34.7E-1"
  - "7."
  - ".7"
  - "999.99e99"
- Invalid numbers
  - "abc"
  - "-e9"
  - "1e"
  - "+"
  - "17.9A"
  - "0.38+"
  - "."
  - "38.38f9"

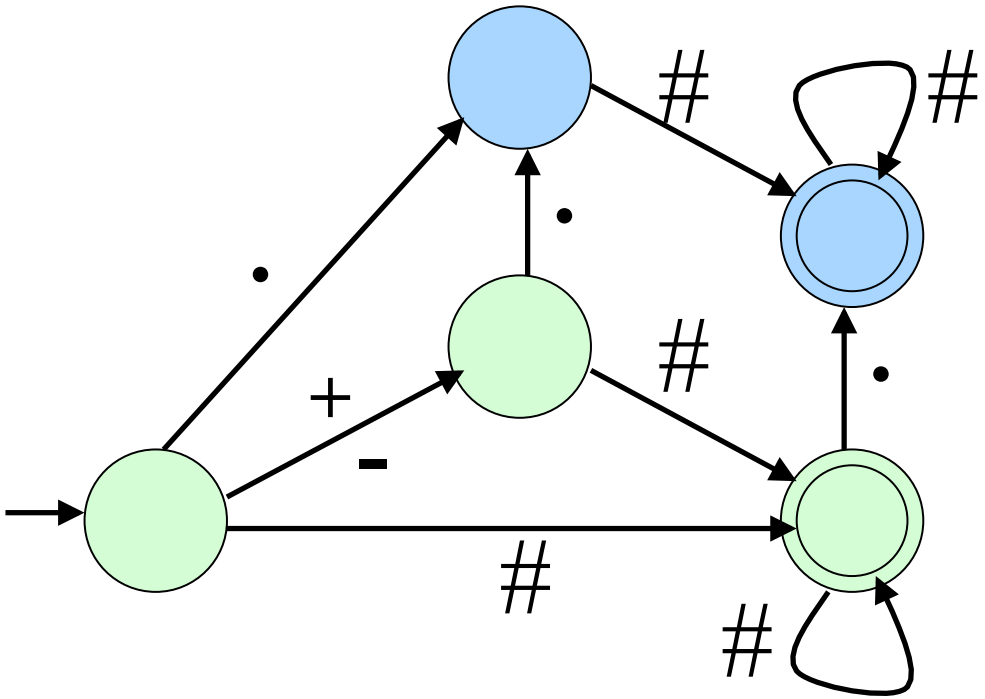
4. Deterministic Finite Automata

- Optional “+” or “-”
- Zero or more digits



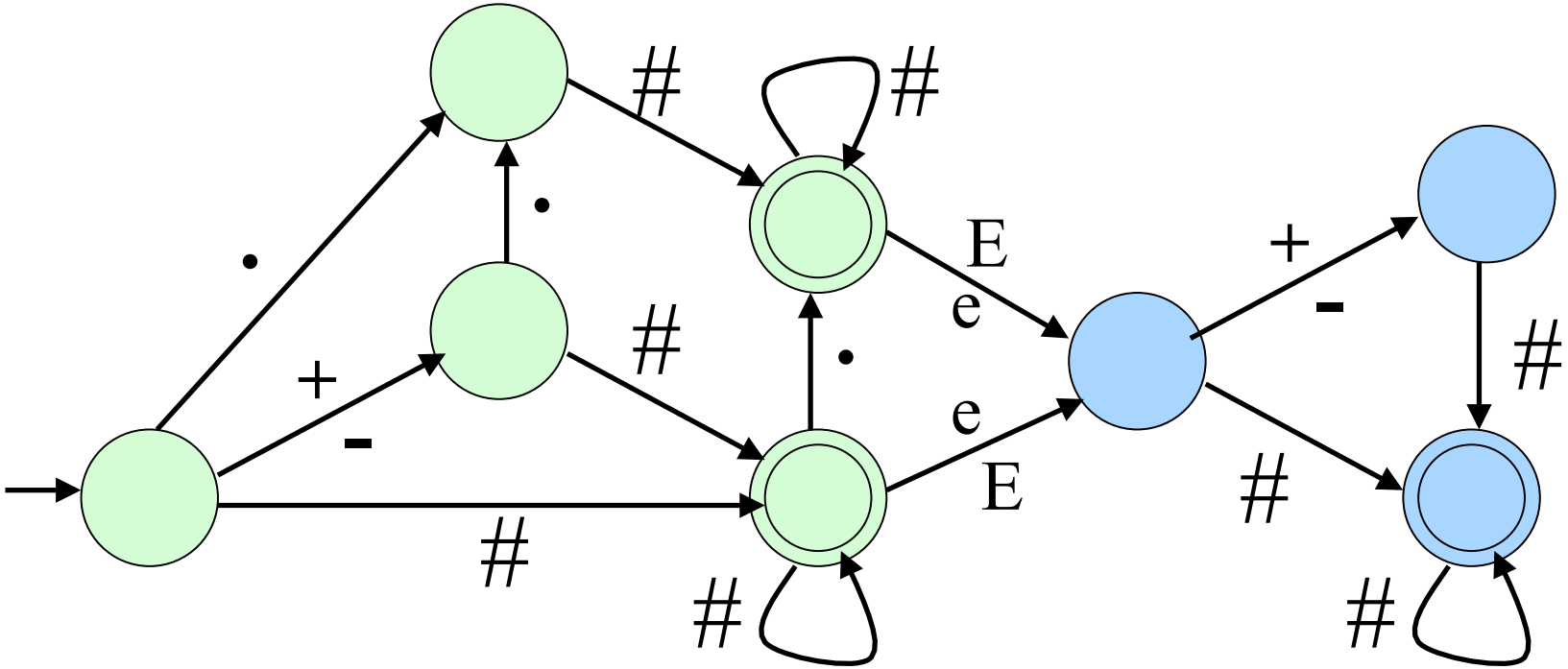
4. Deterministic Finite Automata

- Optional “+” or “-”
- Zero or more digits
- Optional decimal point
  - Followed by zero or more digits



4. Deterministic Finite Automata

- Optional "+" or "-"
- Zero or more digits
- Optional decimal point
  - Followed by zero or more digits
- Optional exponent "E" or "e"
  - Followed by optional "+" or "-"
  - Followed by one or more digits



Summary

- Examples illustrating C
  - Overall program structure
  - Control statements (**if**, **while**, **for**, and **switch**)
  - Character input/output (**getchar()** and **putchar()**)
- Deterministic finite state automata (i.e., state machines)
- Expectations for programming assignments

The Design of C: A Rational Reconstruction

Jennifer Rexford

Goals of this Lecture

- Number systems
  - Binary numbers
  - Finite precision
  - Binary arithmetic
  - Logical operators
- Design rationale for C
  - Decisions **available to** the designers of C
  - Decisions **made by** the designers of C

Number Systems

Why Bits (Binary Digits)?

- Computers are built using digital circuits
  - Inputs and outputs can have only two values
  - True (high voltage) or false (low voltage)
  - Represented as 1 and 0
- Can represent many kinds of information
  - Boolean (true or false)
  - Numbers (23, 79, …)
  - Characters (‘a’, ‘z’, …)
  - Pixels, sounds
  - Internet addresses
- Can manipulate in many ways
  - Read and write
  - Logical operations
  - Arithmetic

Base 10 and Base 2

- Decimal (base 10)
  - Each digit represents a power of 10
  - **4173** = **4** x 10^3 + **1** x 10^2 + **7** x 10^1 + **3** x 10^0
- Binary (base 2)
  - Each bit represents a power of 2
  - **10110** = **1** x 2^4 + **0** x 2^3 + **1** x 2^2 + **1** x 2^1 + **0** x 2^0 = 22

Convert decimal to binary: divide by 2, keep remainders

```
12/2 = 6    R = 0
6/2  = 3    R = 0
3/2  = 1    R = 1
1/2  = 0    R = 1
Result = 1100
```

Writing Bits is Tedious for People

- Octal (base 8)
  - Digits 0, 1, …, 7
- Hexadecimal (base 16)
  - Digits 0, 1, …, 9, A, B, C, D, E, F

0000 = 0	**1000 = 8**
0001 = 1	**1001 = 9**
0010 = 2	**1010 = A**
0011 = 3	**1011 = B**
0100 = 4	**1100 = C**
0101 = 5	**1101 = D**
0110 = 6	**1110 = E**
0111 = 7	**1111 = F**

Thus the 16-bit binary number

1011 0010 1010 1001

converted to hex is

B2A9

Representing Colors: RGB

- Three primary colors
  - Red
  - Green
  - Blue
- Strength
  - 8-bit number for each color (e.g., two hex digits)
  - So, 24 bits to specify a color
- In HTML, e.g. course "Schedule" Web page
  - Red: De-Comment Assignment Due
  - Blue: Reading Period
- Same thing in digital cameras
  - Each pixel is a mixture of red, green, and blue

Finite Representation of Integers

- Fixed number of bits in memory
  - Usually 8, 16, or 32 bits
  - (1, 2, or 4 bytes)
- Unsigned integer
  - No sign bit
  - Always 0 or a positive number
  - All arithmetic is modulo 2^n
- Examples of unsigned integers
  - 00000001 ➔ 1
  - 00001111 ➔ 15
  - 00010000 ➔ 16
  - 00100001 ➔ 33
  - 11111111 ➔ 255

Adding Two Integers

- From right to left, we add each pair of digits
- We write the sum, and add the carry to the next column

Base 10

	1	9	8
+	2	6	4
Sum	4	6	2
Carry	0	1	1

Base 2

	0	1	1
+	0	0	1
Sum	1	0	0
Carry	0	1	1

Binary Sums and Carries

a	b	Sum
0	0	0
0	1	1
1	0	1
1	1	0

XOR
("exclusive OR")

a	b	Carry
0	0	0
0	1	0
1	0	0
1	1	1

AND

```
  0100 0101   <-- 69
+ 0110 0111   <-- 103
-------------
  1010 1100   <-- 172
```

Modulo Arithmetic

- Consider only numbers in a range
  - E.g., five-digit car odometer: 0, 1, …, 99999
  - E.g., eight-bit numbers 0, 1, …, 255
- Roll-over when you run out of space
  - E.g., car odometer goes from 99999 to 0, 1, …
  - E.g., eight-bit number goes from 255 to 0, 1, …
- Adding 2^n doesn't change the answer
  - For eight-bit number, n=8 and 2^n=256
  - E.g., (37 + 256) mod 256 is simply 37
- This can help us do subtraction…
  - a – b: equals a + (256 -1 - b) + 1

One's and Two's Complement

- One's complement: flip every bit
  - E.g., b is 01000101 (i.e., 69 in decimal)
  - One's complement is 10111010
  - That's simply 255-69
- Subtracting from 11111111 is easy (no carry needed!)

```
  1111 1111
- 0100 0101   <-- b
-------------
  1011 1010   <-- one's complement
```

- Two's complement
  - Add 1 to the one's complement
  - E.g., (255 – 69) + 1 ➔ 1011 1011

Putting it All Together

- Computing "a – b"
  - Same as "a + 256 – b"
  - Same as "a + (255 – b) + 1"
  - Same as "a + onesComplement(b) + 1"
  - Same as "a + twosComplement(b)"
- Example: 172 – 69
  - The original number 69: 0100 0101
  - One's complement of 69: 1011 1010
  - Two's complement of 69: 1011 1011
  - Add to the number 172: 1010 1100
  - The sum comes to: 0110 0111
  - Equals: **103** in decimal

```
  1011 1011
+ 1010 1100
-----------
 10110 0111
```

Signed Integers

- Sign-magnitude representation
  - Use one bit to store the sign
    - Zero for positive number
    - One for negative number
  - Examples
    - E.g., 0010 1100 ➔ 44
    - E.g., 1010 1100 ➔ -44
  - Hard to do arithmetic this way, so it is rarely used
- Complement representation
  - One's complement
    - Flip every bit
    - E.g., 1101 0011 ➔ -44
  - Two's complement
    - Flip every bit, then add 1
    - E.g., 1101 0100 ➔ -44

Overflow: Running Out of Room

- Adding two large integers together
  - Sum might be too big for the number of bits available
  - What happens?
- Unsigned integers
  - All arithmetic is “modulo” arithmetic
  - Sum would just wrap around
- Signed integers
  - Can get nonsense values
  - Example with 16-bit integers
    - Sum: 10000+20000+30000
    - Result: -5536

Bitwise Operators: AND and OR

- Bitwise AND (&)

&	0	1
0	0	0
1	0	1

  - Mod on the cheap!
    - E.g., 53 % 16
    - … is same as 53 & 15;

53	0	0	1	1	0	1	0	1
& 15	0	0	0	0	1	1	1	1
5	0	0	0	0	0	1	0	1

- Bitwise OR (|)

|	0	1
0	0	1
1	1	1

Bitwise Operators: Not and XOR

- One's complement (~)
  - Turns 0 to 1, and 1 to 0
  - E.g., set last three bits to 0
    - x = x & ~7;
- XOR (^)
  - 0 if both bits are the same
  - 1 if the two bits are different

^	0	1
0	0	1
1	1	0

Bitwise Operators: Shift Left/Right

- Shift left (<<): Multiply by powers of 2
  - Shift some # of bits to the left, filling the blanks with 0

53	0	0	1	1	0	1	0	1
53<<2	1	1	0	1	0	0	0	0

- Shift right (>>): Divide by powers of 2
  - Shift some # of bits to the right
  - For unsigned integer, fill in blanks with 0
  - What about signed negative integers?
    - Can vary from one machine to another!

53	0	0	1	1	0	1	0	1
53>>2	0	0	0	0	1	1	0	1

Example: Counting the 1's

- How many 1 bits in a number?
  - E.g., how many 1 bits in the binary representation of 53?

0	0	1	1	0	1	0	1

  - Four 1 bits
- How to count them?
  - Look at one bit at a time
  - Check if that bit is a 1
  - Increment counter
- How to look at one bit at a time?
  - Look at the last bit: n & 1
  - Check if it is a 1: (n & 1) == 1, or simply (n & 1)

Counting the Number of '1' Bits

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
   unsigned int n;
   unsigned int count;
   printf("Number: ");
   if (scanf("%u", &n) != 1) {
      fprintf(stderr, "Error: Expect unsigned int.\n");
      exit(EXIT_FAILURE);
   }
   for (count = 0; n > 0; n >>= 1)
       count += (n & 1);
   printf("Number of 1 bits: %u\n", count);
   return 0;
}
```

Summary

- Computer represents everything in binary
  - Integers, floating-point numbers, characters, addresses, …
  - Pixels, sounds, colors, etc.
- Binary arithmetic through logic operations
  - Sum (XOR) and Carry (AND)
  - Two's complement for subtraction
- Bitwise operators
  - AND, OR, NOT, and XOR
  - Shift left and shift right
  - Useful for efficient and concise code, though sometimes cryptic

The Design of C

Goals of C

Designers wanted C to support:

- **Systems programming**
  - Development of Unix OS
  - Development of Unix programming tools

But also:

- **Applications programming**
  - Development of financial, scientific, etc. applications

Systems programming was the primary intended use

The Goals of C (cont.)

The designers of wanted C to be:

- Low-level
  - Close to assembly/machine language
  - Close to hardware

But also:

- Portable
  - Yield systems software that is easy to port to differing hardware

The Goals of C (cont.)

The designers wanted C to be:

- Easy for **people** to handle
  - Easy to understand
  - **Expressive**
    - High (functionality/sourceCodeSize) ratio

But also:

- Easy for **computers** to handle
  - Easy/fast to compile
  - Yield efficient machine language code

Commonality:

- Small/simple

Design Decisions

In light of those goals…

- What design decisions did the designers of C **have**?
- What design decisions did they **make**?

Consider programming language features,

from simple to complex…

Feature 1: Data Types

- Previously in this lecture:
  - Bits can be combined into bytes
  - Our interpretation of a collection of bytes gives it meaning
    - A signed integer, an unsigned integer, a RGB color, etc.
- **Data type:** well-defined interpretation of collection of bytes
- A high-level language should provide primitive data types
  - Facilitates abstraction
  - Facilitates manipulation via associated well-defined operators
  - Enables compiler to check for mixed types, inappropriate use of types, etc.

Primitive Data Types

- Thought process
  - C should handle:
    - **Integers**
    - **Characters**
    - Character **strings**
    - **Logical** (alias **Boolean**) data
    - **Floating-point** numbers
  - C should be small/simple
- Decisions
  - Provide **integer**, **character**, and **floating-point** data types
  - **Do not** provide a character **string** data type (More on that later)
  - **Do not** provide a **logical** data type (More on that later)

Integer Data Types

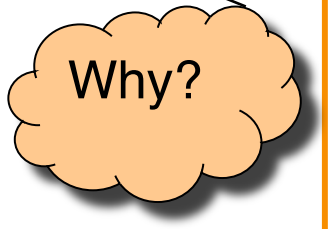
- Thought process
  - For flexibility, should provide integer data types of various sizes
  - For portability at **application** level, should specify size of each data type
  - For portability at **systems** level, should define integral data types in terms of **natural word size** of computer
  - Primary use will be **systems** programming

Integer Data Types (cont.)

- Decisions
  - Provide three integer data types: **`short`**, **`int`**, and **`long`**
  - Do *not* specify sizes; instead:
    - **`int`** is natural word size
    - 2 <= bytes in **`short`** <= bytes in **`int`** <= bytes in **`long`**
- Incidentally, on hats using gcc217
  - Natural word size: 4 bytes
  - **`short`**: 2 bytes
  - **`int`**: 4 bytes
  - **`long`**: 4 bytes

Integer Constants

- Thought process
  - People naturally use decimal
  - Systems programmers often use binary, octal, hexadecimal
- Decisions
  - Use decimal notation as default
  - Use "0" prefix to indicate octal notation
  - Use "0x" prefix to indicate hexadecimal notation
  - Do not allow binary notation; too verbose, error prone
  - Use "L" suffix to indicate `long` constant
  - Do not use a suffix to indicate `short` constant; instead must use cast
- Examples
  - `int`: `123`, `-123`, `0173`, `0x7B`
  - `long`: `123L`, `-123L`, `0173L`, `0x7BL`
  - `short`: `(short)123`, `(short)-123`, `(short)0173`, `(short)0x7B`



Unsigned Integer Data Types

- Thought process
  - Must represent positive and negative integers
    - Signed types are essential
  - Unsigned data can be twice as large as signed data
    - Unsigned data could be useful
  - Unsigned data are good for bit-level operations
    - Bit-level operations are common in systems programming
  - Implementing both signed and unsigned data types is complex
    - Must define behavior when an expression involves both

Unsigned Integer Data Types (cont.)

- Decisions
  - Provide unsigned integer types: **`unsigned short`**, **`unsigned int`**, and **`unsigned long`**
  - Conversion rules in mixed-type expressions are complex
    - Generally, mixing signed and unsigned converts signed to unsigned
    - See King book Section 7.4 for details

Was providing unsigned types a good decision?

Do you see any potential problems?

What decision did the designers of Java make?

Unsigned Integer Constants

- Thought process
  - "L" suffix distinguishes **`long`** from **`int`**; also could use a suffix to distinguish signed from unsigned
  - Octal or hexadecimal probably are used with bit-level operators
- Decisions
  - Default is signed
  - Use "U" suffix to indicate unsigned
  - Integers expressed in octal or hexadecimal automatically are unsigned
- Examples
  - **`unsigned int`**: **`123U`**, **`0173`**, **`0x7B`**
  - **`unsigned long`**: **`123UL`**, **`0173L`**, **`0x7BL`**
  - **`unsigned short`**: **`(short)123U`**, **`(short)0173`**, **`(short)0x7B`**

There's More!

To be continued next lecture!

The Design of C: A Rational Reconstruction (cont.)

Jennifer Rexford

Goals of this Lecture

- Help you learn about:
  - The decisions that were **available to** the designers of C
  - The decisions that were **made by** the designers of C
- Why?
  - Learning the design rationale of the C language provides a richer understanding of C itself
  - A power programmer knows both the programming language and its design rationale

Character Data Types

- Thought process
  - The most common character codes are (were!) ASCII and EBCDIC
  - ASCII is 7-bit
  - EBCDIC is 8-bit
- Decisions
  - Provide type **`char`**
  - Type **`char`** should be one byte

Character Data Types (cont.)

- Tangential Decision
  - **`char`** should be an integer type
    - Can use type **`char`** to store small integers
    - Can do arithmetic with data of type **`char`**
    - Can freely mix **`char`** and integer data
      - **`('a' + 1)`** is **`'b'`** (assuming ASCII)
      - **`('0' + 5)`** is **`'5'`** (assuming ASCII)

How does Java handle these expressions?

Was that a good decision?

Character Constants

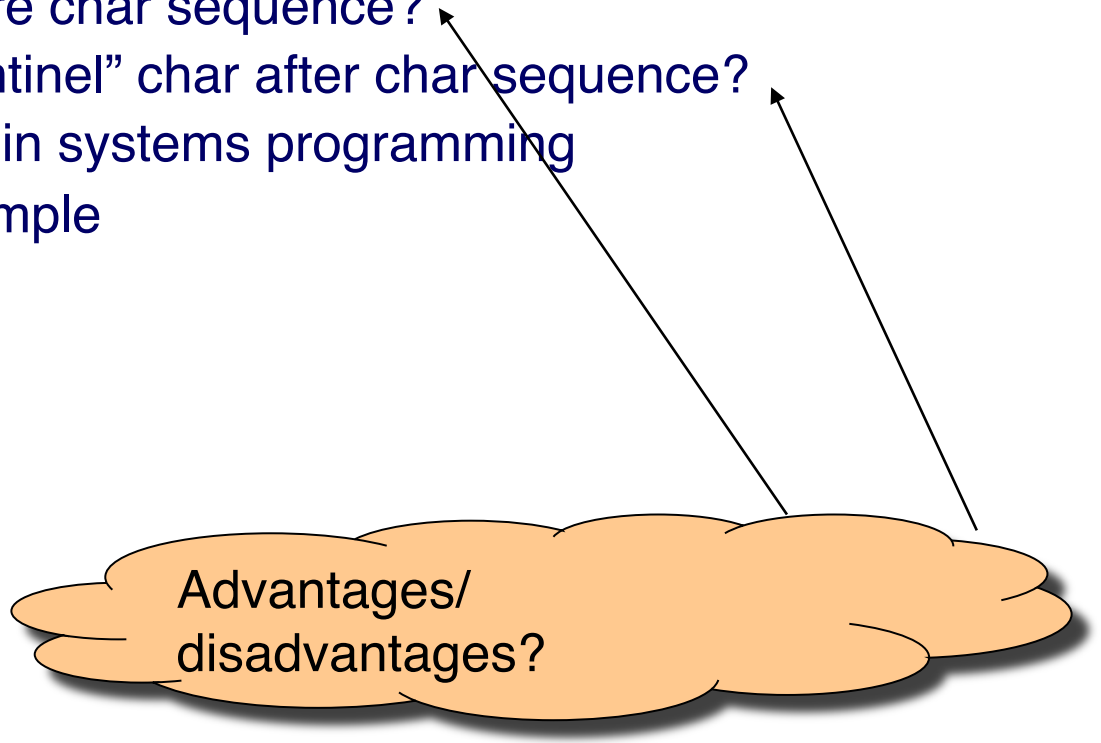
- Thought process
  - Could represent character constants as `int` constants, with truncation of high-order bytes
  - More readable to use single quote (`'a'`, `'b'`, etc.); but then…
  - Need special way to represent the single quote character
  - Need special ways to represent non-printable characters (e.g. newline, tab, space, etc.)
- Decisions
  - Provide single quote syntax
  - Use backslash to express special characters

Character Constants (cont.)

- Examples
  - `'a'` **the a character**
  - `(char)97` the a character
  - `(char)0141` the a character
  - `'\o141'` the a character, octal character form
  - `'\x61'` the a character, hexadecimal character form
  - `'\0'` **the null character**
  - `'\a'` bell
  - `'\b'` backspace
  - `'\f'` formfeed
  - `'\n'` **newline**
  - `'\r'` carriage return
  - `'\t'` **horizontal tab**
  - `'\v'` vertical tab
  - `'\\'` backslash
  - `'\''` single quote

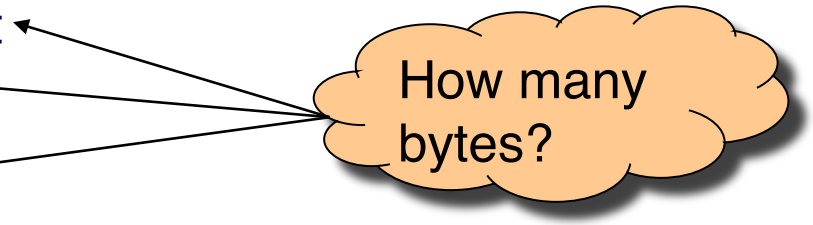
Strings

- Thought process
  - String can be represented as a sequence of chars
  - How to know where char sequence ends?
    - Store length before char sequence?
    - Store special “sentinel” char after char sequence?
  - Strings are common in systems programming
  - C should be small/simple



Strings (cont.)

- Decisions
  - Adopt a convention
    - String consists of a sequence of chars terminated with the null (`'\0'`) character
  - Use double-quote syntax (e.g. `"abc"`, `"hello"`) to represent a string constant
  - Provide no other language features for handling strings
    - Delegate string handling to standard library functions
- Examples
  - `"abc"` is a string constant
  - `'a'` is a `char` constant
  - `"a"` is a string constant

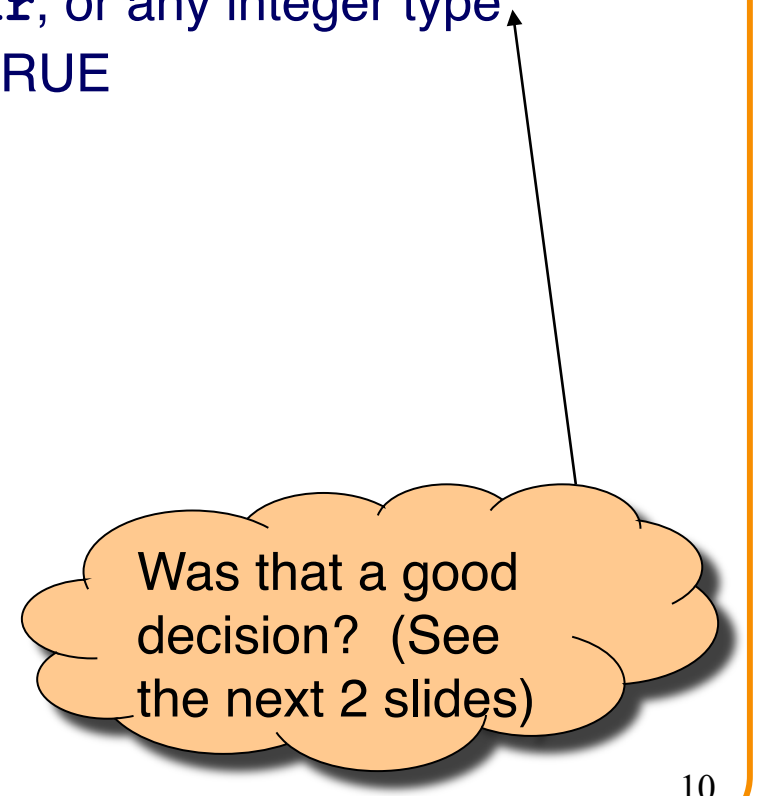


Logical Data Type

- Thought process
  - Representing a logical value (TRUE or FALSE) requires only one **bit**
  - Smallest entity that can be addressed is one **byte**
  - Type **`char`** is one byte, so could be used to represent logical values
  - C should be small/simple

Logical Data Type (cont.)

- Decisions
  - Don't define a logical data type
  - Represent logical data using type **`char`**, or any integer type
  - Convention: 0 => FALSE, non-0 => TRUE
  - Convention used by:
    - Relational operators (**`<`**, **`>`**, etc.)
    - Logical operators (**`!`**, **`&&`**, **`||`**)
    - Statements (**`if`**, **`while`**, etc.)



Logical Data Type (cont.)

- Note
  - Using integer data to represent logical data permits shortcuts

```
…
int i;
…
if (i)  /* same as (i != 0) */
   statement1;
else
   statement2;
…
```

Are such shortcuts beneficial?

Logical Data Type (cont.)

- Note
  - The lack of logical data type cripples compiler's ability to detect some errors

```
…
int i;
…
i = 0;
…
if (i = 5)
    statement1;
else
    statement2;
…
```

What is the problem with this code?

What is the effect of this code?

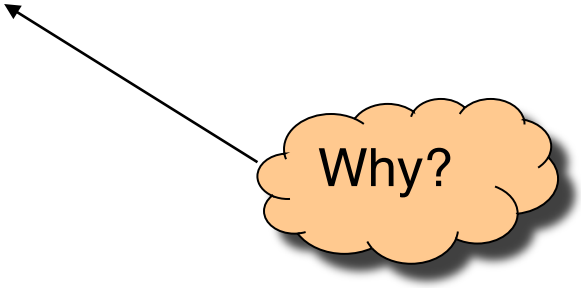
How does Java handle this code?

Floating-Point Data Types

- Thought process
  - Systems programs use floating-point data infrequently
  - But some application domains (e.g. scientific) use floating-point data often
- Decisions
  - Provide three floating-point data types: **`float`**, **`double`**, and **`long double`**
  - bytes in **`float`** <= bytes in **`double`** <= bytes in **`long double`**
- Incidentally, on hats using gcc217
  - **`float`**: 4 bytes
  - **`double`**: 8 bytes
  - **`long double`**: 12 bytes

Floating-Point Constants

- Thought process
  - Convenient to allow both fixed-point and scientific notation
  - Decimal is sufficient; no need for octal or hexadecimal
- Decisions
  - Any constant that contains decimal point or "E" is floating-point
  - The default floating-point type is **`double`**
  - Append "**`F`**" to indicate **`float`**
  - Append "**`L`**" to indicate **`long double`**
- Examples
  - **`double`**: **`123.456`**, **`1E-2`**, **`-1.23456E4`**
  - **`float`**: **`123.456F`**, **`1E-2F`**, **`-1.23456E4F`**
  - **`long double`**: **`123.456L`**, **`1E-2L`**, **`-1.23456E4L`**



Feature 2: Operators

- A high-level programming language should have **operators**
- Operators combine with constants and variables to form expressions

Kinds of Operators

- Thought process
  - Should handle typical operations
  - Should handle bit-level programming ("bit fiddling")
- Decisions
  - Provide typical arithmetic operators: `+ - * / %`
  - Provide typical relational operators: `== != < <= > >=`
    - Each evaluates to 0=>FALSE or 1=>TRUE
  - Provide typical logical operators: `! && ||`
    - Each interprets 0=>FALSE, non-0=>TRUE
    - Each evaluates to 0=>FALSE or 1=>TRUE
  - Provide bitwise operators: `~ & | ^ >> <<`
  - Provide a cast operator: `(type)`

Assignment Operator

- Thought process
  - Must have a way to assign a value to a variable
  - Many high-level languages provide an assignment **statement**
  - Would be more expressive to define an assignment **operator**
    - Performs assignment, and then evaluates to the assigned value
    - Allows expressions that involve assignment to appear within larger expressions
- Decisions
  - Provide assignment operator: `=`
  - Define assignment operator so it changes the value of a variable, and also evaluates to that value

Assignment Operator (cont.)

- Examples

```
i = 0;
  /* Assign 0 to i. Evaluate to 0.
     Discard the 0. */

i = j = 0;
  /* Assign 0 to j. Evaluate to 0.
     Assign 0 to i. Evaluate to 0.
     Discard the 0. */

while ((i = getchar()) != EOF) …
   /* Read a character.  Assign it to i.
      Evaluate to that character.
      Compare that character to EOF.
      Evaluate to 0 (FALSE) or 1 (TRUE). */
```

Does the expressiveness affect clarity?

Increment and Decrement Operators

- Thought process
  - The construct `i = i + 1` is common
  - Special purpose increment and decrement operators would make code more expressive
  - Such operators would complicate the language and compiler
- Decisions
  - The convenience outweighs the complication
  - Provide increment and decrement operators: `++ --`

Special-Purpose Assignment Operators

- Thought process
  - Constructs such as `i = i + n` and `i = i * n` are common.
  - Special-purpose assignment operators would make code more expressive
  - Such operators would complicate the language and compiler
- Decisions
  - The convenience outweighs the complication
  - Provide special-purpose assignment operators: `+= -= *= /= ~= &= |= ^= <<= >>=`

Sizeof Operator

- Thought process
  - The sizes of most primitive types are unspecified
  - C must provide a way to determine the size of a given data type programmatically
- Decisions
  - Provide a **`sizeof`** operator
    - Applied at compile-time
    - Operand can be a **data type**
    - Operand can be an **expression**, from which the compiler infers a data type
- Examples, on hats using gcc217
  - **`sizeof(int)`** evaluates to 4
  - **`sizeof(i)`** evaluates to 4 (where **`i`** is a variable of type **`int`**)
  - **`sizeof(i+1)`** evaluates to 4 (where **`i`** is a variable of type **`int`**)

Other Operators

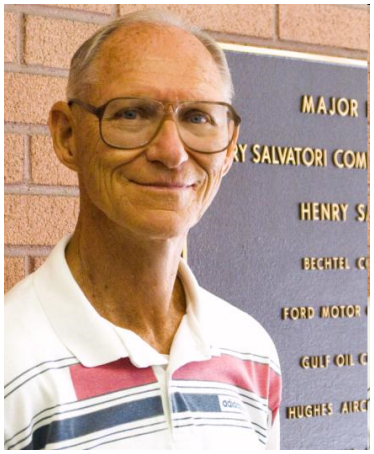
- Function call operator
  - Should mimic the familiar mathematical notation
  - `function(param1, param2, …)`
- Conditional operator: `?:`
  - The only ternary operator
  - See King book
- Sequence operator: `,`
  - See King book
- Pointer-related operators: `& *`
  - Described later in the course
- Structure-related operators (. ->)
  - Described later in the course

Feature 3: Control Statements

- A programming language must provide **statements**
- Some statements must affect flow of control

Control Statements

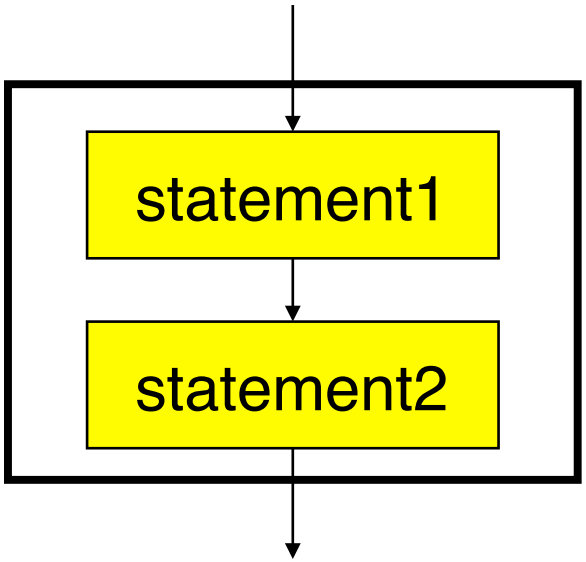
- Thought process
  - **Boehm** and **Jacopini** proved that any algorithm can be expressed as the nesting of only 3 control structures:



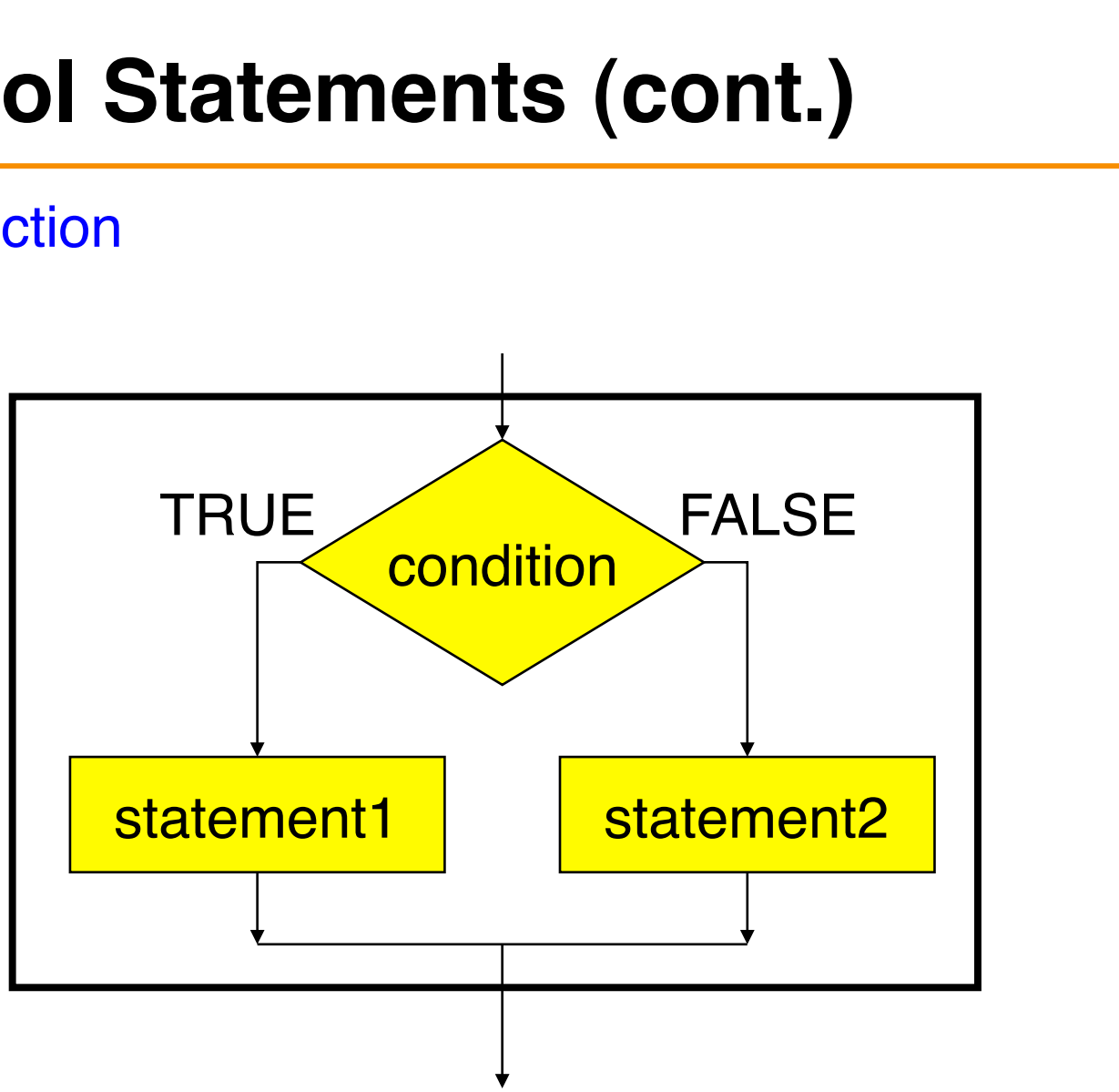
Barry Boehm

Control Statements (cont.)

(1) Sequence



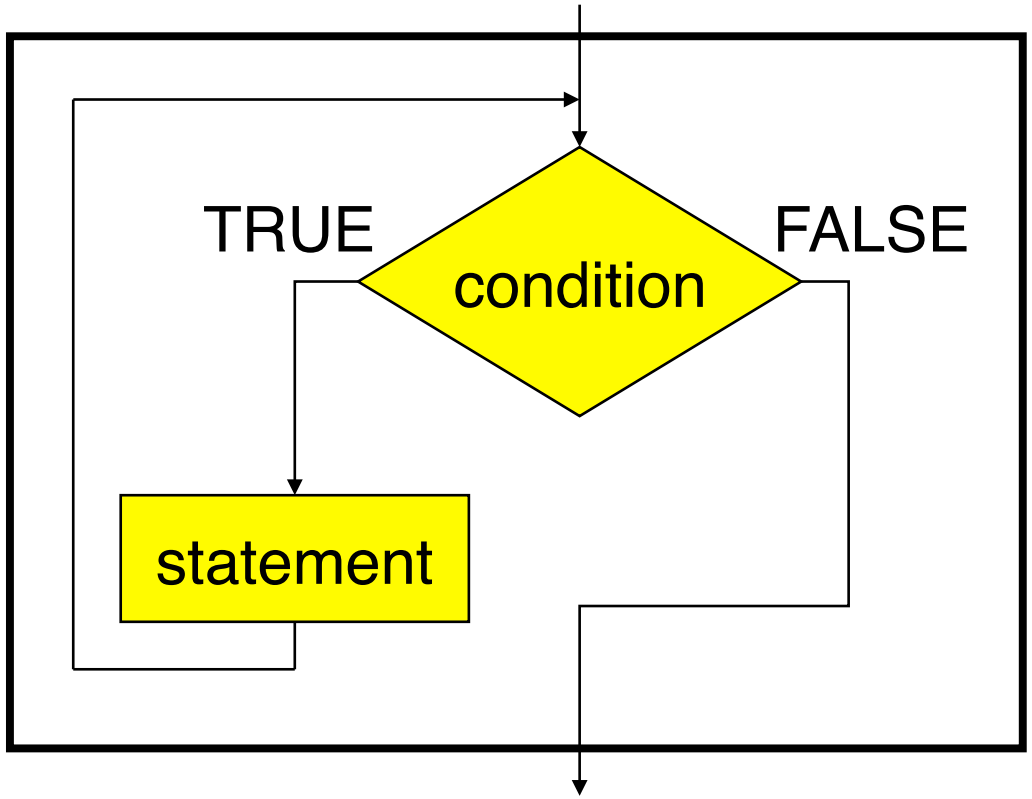
Control Statements (cont.)



(2) Selection

Control Statements (cont.)

(3) Repetition



Control Statements (cont.)

- Thought Process (cont.)
  - **Dijkstra** argued that any algorithm **should** be expressed using *only* those three control structures (*GOTO Statement Considered Harmful* paper)
  - The ALGOL programming language implemented control statements accordingly

Edsgar Dijkstra

- Decisions
  - Provide statements to implement those 3 control structures
  - For convenience, provide a few extras

Sequence Statement

- Decision
  - **Compound** statement, alias **block**

```
{
    statement1;
    statement2;
     …
}
```

Selection Statements

- Decisions
  - **if** statement, for one-path or two-path decisions

```
if (integerExpr)
   statement1;
```

```
if (integerExpr)
   statement1;
else
   statement2;
```

Selection Statements (cont.)

- Decisions (cont.)
  - **switch** and **break** statements, for multi-path decisions

```
switch (integerExpr) {
    case integerConstant1:
        …
        break;
    case integerConstant2:
        …
        break;
    …
    default:
        …
}
```

What if these **break** statements are omitted?

Was that use of **break** a good design decision?

Repetition Statements

- Decisions
  - **while** statement, for general repetition

```
while (integerExpr)
   statement;
```

  - **for** statement, for counting loops

```
for (initialExpr; integerExpr; incrementExpr)
   statement;
```

  - **do…while** statement, for loops with test at trailing edge

```
do
   statement;
while (integerExpr);
```

Other Control Statements

- Decisions
  - **break** statement (revisited)
    - Breaks out of closest enclosing **`switch`** **or repetition** statement
  - **continue** statement
    - Skips remainder of current loop iteration
    - Continues with next loop iteration
    - Can be difficult to understand; generally should avoid
  - **goto** statement and labels
    - Avoid!!! (as per Dijkstra)

Feature 4: Input/Output

- A programming language must provide facilities for reading and writing data
- Alternative: A programming **environment** must provide such facilities

Input/Output Facilities

- Thought process
  - Unix provides the stream abstraction
    - A stream is a sequence of characters
  - Unix provides 3 standard streams
    - Standard input, standard output, standard error
  - C should be able to use those streams, and others
  - I/O facilities are complex
  - C should be small/simple
- Decisions
  - **Do not** provide I/O facilities in C
  - Instead provide a **standard library** containing I/O facilities
    - Constants: **`EOF`**
    - Data types: **`FILE`** (described later in course)
    - Variables: **`stdin`**, **`stdout`**, and **`stderr`**
    - Functions: …

Reading Characters

- Thought process
  - Need function to read a single character from **`stdin`**
  - Function must have a way to indicate failure, that is, to indicate that no characters remain
- Decisions
  - Provide **`getchar()`** function
  - Make return type of **`getchar()`** wider than **`char`**
    - Make it **`int`**; that's the natural word size
  - Define **`getchar()`** to return **`EOF`** (a special non-character **`int`**) to indicate failure
- Note
  - There is no such thing as "the **`EOF`** character"

Writing Characters

- Thought process
  - Need function to write a single character to **`stdout`**
- Decisions
  - Provide a **`putchar()`** function
  - Define **`putchar()`** to accept one parameter
    - For symmetry with **`getchar()`**, parameter should be an **`int`**

Reading Other Data Types

- Thought process
  - Must convert external form (sequence of character codes) to internal form
  - Could provide **`getshort()`**, **`getint()`**, **`getfloat()`**, etc.
  - Could provide one parameterized function to read any primitive type of data
- Decisions
  - Provide **`scanf()`** function
  - Can read any primitive type of data
  - First parameter is a **format string** containing **conversion specifications**
- See King book for details

Writing Other Data Types

- Thought process
  - Must convert internal form to external form (sequence of character codes)
  - Could provide **`putshort()`**, **`putint()`**, **`putfloat()`**, etc.
  - Could provide one parameterized function to write any primitive type of data
- Decisions
  - Provide **`printf()`** function
  - Can write any primitive type of data
  - First parameter is a **format string** containing **conversion specifications**
- See King book for details

Other I/O Facilities

- Decisions
  - **`fopen()`**: Open a stream
  - **`fclose()`**: Close a stream
  - **`fgetc()`**: Read a character from specified stream
  - **`fputc()`**: Write a character to specified stream
  - **`fgets()`**: Read a line/string from specified stream
  - **`fputs()`**: Write a line/string to specified stream
  - **`fscanf()`**: Read data from specified stream
  - **`fprintf()`**: Write data to specified stream
- Described in King book, and later in the course after covering files, arrays, and strings

Summary

- C's design goals affected decisions concerning language features:
  - Data types
  - Operators
  - Control statements
  - I/O facilities
- Knowing the design goals and how they affected the design decisions can yield a rich understanding of C

Testing

Jennifer Rexford

The material for this lecture is drawn, in part, from
The Practice of Programming (Kernighan & Pike) Chapter 6

Quotations on Program Testing

"On two occasions I have been asked [by members of Parliament!], 'Pray, Mr. Babbage, *if you put into the machine wrong figures, will the right answers come out*?' I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question."

- Charles Babbage

"Program testing can be quite effective for showing the *presence* of bugs, but is hopelessly inadequate for showing their *absence*."

- Edsger Dijkstra

"Beware of bugs in the above code; I have only *proved* it correct, not *tried* it."

- Donald Knuth

Goals of This Lecture

- Help you learn about:
  - Internal testing
  - External testing
  - General testing strategies
- Why?
  - It's hard to know if a large program works properly
  - A power programmer expends ***at least*** **as much effort writing test code** as writing the program itself

Program Verification

- **Ideally**: Prove that your program is correct
  - Can you **prove** properties of the program?
  - Can you **prove** that it even terminates?!!!
    - See Turing's "Halting Problem"

Program Testing

- **Pragmatically**: Convince yourself that your program *probably* works

External vs. Internal Testing

- **External** testing
  - Designing data to test your program
- **Internal** testing
  - Designing your program to test itself

External Testing: Statement Testing

(1) Statement testing

- "Testing to satisfy the criterion that each statement in a program be executed at least once during program testing."
  - Glossary of Computerized System and Software Development Terminology

Statement Testing Example

- Example pseudocode:

```
if (condition1)
    statement1;
else
    statement2;
…
if (condition2)
    statement3;
else
    statement4;
…
```

Statement testing:

Should make sure both “if” statements and all 4 nested statements are executed

How many data sets are required?

External Testing: Path Testing

(2) Path testing

- "Testing to satisfy coverage criteria that each logical path through the program be tested. Often paths through the program are grouped into a finite set of classes. One path from each class is then tested."
  - Glossary of Computerized System and Software Development Terminology

- More difficult than statement testing
  - For simple programs, can enumerate all paths through the code
  - Otherwise, sample paths through code with random input

Path Testing Example

- Example pseudocode:

```
if (condition1)
    statement1;
else
    statement2;
…
if (condition2)
    statement3;
else
    statement4;
…
```

Path testing:

Should make sure all logical paths are executed

How many data sets are required?

- Realistic program => combinatorial explosion!!!

External Testing: Boundary Testing

(3) Boundary testing

- "A testing technique using input values at, *just below*, and *just above*, the defined limits of an input domain; and with input values causing outputs to be at, just below, and just above, the defined limits of an output domain."

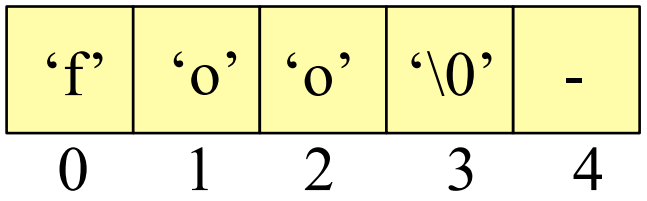
  - Glossary of Computerized System and Software Development Terminology

- Alias **corner case** testing

Boundary Testing Example

- Specification:
  - Read line from **`stdin`**, store as string in array (without '\n')
- First attempt:

```
int i;
char s[ARRAYSIZE];
for (i=0; ((i < ARRAYSIZE-1) && (s[i]=getchar()) != '\n'); i++)
   ;
s[i] = '\0';
```



ARRAYSIZE = 5

Example Boundary Conditions

- Consider boundary conditions:
  1. stdin contains no characters (empty file)
  2. stdin starts with '\n' (empty line)
  3. stdin contains characters but no '\n‘
  4. stdin line contains exactly ARRAYSIZE-1 characters
  5. stdin line contains exactly ARRAYSIZE characters
  6. stdin line contains more than ARRAYSIZE characters

Testing the First Attempt

- Embed code in complete program:

```
#include <stdio.h>
enum {ARRAYSIZE = 5}; /* Artificially small */
int main(void)
{
   int i;
   char s[ARRAYSIZE];
   for (i=0; ((i < ARRAYSIZE-1) && (s[i]=getchar()) != '\n'); i++)
      ;
   s[i] = '\0';
   for (i = 0; i < ARRAYSIZE; i++) {
      if (s[i] == '\0') break;
      putchar(s[i]);
   }
   return 0;
}
```

Test Results for First Attempt

```
int i;
char s[ARRAYSIZE];
for (i=0; ((i < ARRAYSIZE-1) && (s[i]=getchar()) != '\n')); i++)
   ;
s[i] = '\0';
```

1. stdin contains no characters (empty file)
   - → ÿÿÿÿÿÿ Fail
2. stdin starts with '\n' (empty line)
   - $_n$ → Pass
3. stdin contains characters but no '\n'
   - ab → abÿÿÿ Fail
4. stdin line contains exactly ARRAYSIZE-1 characters
   - abc_n → abc Pass
5. stdin line contains exactly ARRAYSIZE characters
   - $abcd_n$ → abcd Pass
6. stdin line contains more than ARRAYSIZE characters
   - $abcde_n$ → abcd Pass or Fail???

Again:
Does it work?

Ambiguity in Specification

- If stdin line is too long, what should happen?
  - Keep first ARRAYSIZE characters, discard the rest?
  - Keep first ARRAYSIZE -1 characters + '\0' char, discard the rest?
  - Keep first ARRAYSIZE -1 characters + '\0' char, save the rest for the next call to the input function?
- Specification didn't say what to do if MAXLINE is exceeded
  - Testing has uncovered a design or specification problem!
- Define what to do
  - Keep first ARRAYSIZE -1 characters + '\0' char
  - Save the rest for the next call to the input function

A Second Attempt

- Second attempt:

```
int i;
char s[ARRAYSIZE];
for (i = 0; i < ARRAYSIZE-1; i++) {
   s[i] = getchar();
   if ((s[i] == EOF) || (s[i] == '\n'))
      break;
}
s[i] = '\0';
```

Testing the Second Attempt

- Embed code in complete program:

```
#include <stdio.h>
enum {ARRAYSIZE = 5}; /* Artificially small */
int main(void)
{
   int i;
   char s[ARRAYSIZE];
   for (i = 0; i < ARRAYSIZE-1; i++) {
      s[i] = getchar();
      if ((s[i] == EOF) || (s[i] == '\n'))
         break;
   }
   s[i] = '\0';
   for (i = 0; i < ARRAYSIZE; i++) {
      if (s[i] == '\0') break;
      putchar(s[i]);
   }
   return 0;
}
```

Test Results for Second Attempt

```
int i;
char s[ARRAYSIZE];
for (i = 0; i < ARRAYSIZE-1; i++) {
   s[i] = getchar();
   if ((s[i] == EOF) || (s[i] == '\n'))
      break;
}
s[i] = '\0';
```

1. stdin contains no characters (empty file)
   - → Pass
2. stdin starts with '\n' (empty line)
   - $_n$ → Pass
3. stdin contains characters but no '\n'
   - ab → ab Pass
4. stdin line contains exactly ARRAYSIZE-1 characters
   - abc$_n$ → abc Pass
5. stdin line contains exactly ARRAYSIZE characters
   - abcd$_n$ → abcd Pass
6. stdin line contains more than ARRAYSIZE characters
   - abcde$_n$ → abcd Pass

Again:
Does it work?

Morals of this Little Story

- Testing can reveal the presence of bugs
  - … but *not* their absence
- Complicated boundary cases often are symptomatic of bad design or bad specification
  - Clean up the specification if you can
  - Otherwise, fix the code

External Testing: Stress Testing

(4) Stress testing

- "Testing conducted to evaluate a system or component at or beyond the limits of its specified requirements"
  - Glossary of Computerized System and Software Development Terminology

- What to generate
  - Very large input sets
  - Random input sets (binary vs. ASCII)

- Use computer to generate input sets

Stress Testing Example 1

- Specification: Copy all characters of stdin to stdout
- Attempt:

```
#include <stdio.h>
int main(void) {
   char c;
   while ((c = getchar()) != EOF)
      putchar(c);
   return 0;
}
```

Does it work?
Hint: Consider random input sets

Does this example shed light on the previous one?

Stress Testing Example 2

- Specification: Print number of characters in stdin
- Attempt:

```
#include <stdio.h>
int main(void) {
   char charCount = 0;
   while (getchar() != EOF)
      charCount++;
   printf("%d\n", charCount);
   return 0;
}
```

Does it work?
Hint: Consider large input sets

External Testing Summary

- External testing: Designing data to test your program
- External testing taxonomy
  (1) Statement testing
  (2) Path testing
  (3) Boundary testing
  (4) Stress testing

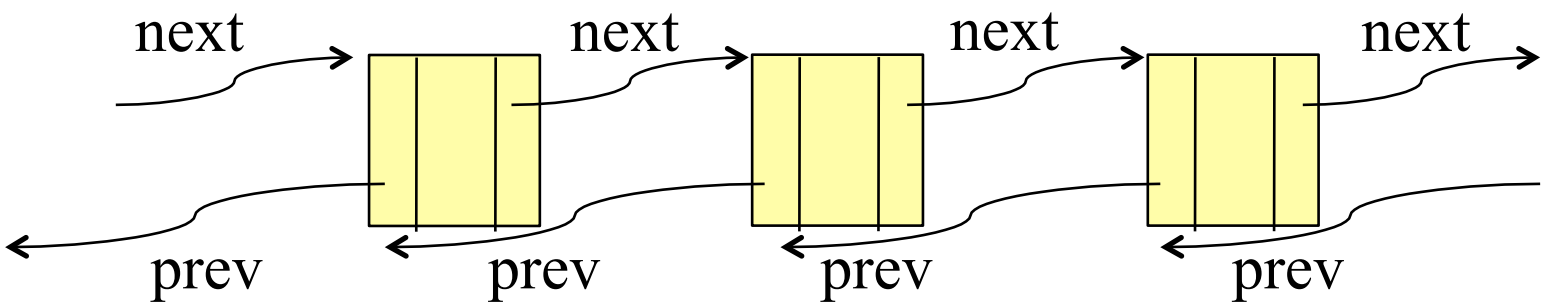
Internal Testing

- Internal testing
  - Designing your program to test itself
  - Four techniques…

Internal Testing: Checking Invariants

(1) Checking invariants

- Function should check aspects of data structures that shouldn't vary
  - Check the data structure at the beginning
  - Check again at the end of the function
- Example: a doubly-linked list



  - Check that the "next" node points back to the "prev" node

Checking Invariants With `assert()`

- The **`assert`** macro
  - One actual parameter
  - Evaluates to 0 (FALSE) or non-0 (TRUE)
- If TRUE:
  - Do nothing
- If FALSE:
  - Print message to stderr "assert at line x failed"
  - Exit the process

```
int isValid(MyType object) {
    …
    Check invariants here.
    Return 1 (TRUE) if object passes
    all tests, and 0 (FALSE) otherwise.
    …
}

void myFunction(MyType object) {
    assert(isValid(object));
    …
    Manipulate object here.
    …
    assert(isValid(object));
}
```

Other Uses of `assert()`

- Validate formal parameters

```
int gcd(int i, int j) {
   assert(i > 0);
   assert(j > 0);
   …
}
```

- Check for "impossible" logical flow

```
switch (state) {
   case START: … break;
   case COMMENT: … break;
   …
   default: assert(0); /* Never should get here */
}
```

Internal Testing: Return Values

(2) Checking function return values

- In C:
  - No exception-handling mechanism
  - Function that detects error typically indicates so via return value
  - Programmer easily can forget to check return value
  - Programmer (generally) **should** check return value

Checking Return Values (cont.)

(2) Checking function return values (cont.)

- Example: **`scanf()`** returns number of values read

Bad code

```
int i;
scanf("%d", &i);
```

Good code

```
int i;
if (scanf("%d", &i) != 1)
   /* Error */
```

- Example: **`printf()`** can fail if writing to file and disk is full; returns number of characters (not values) written

Bad code???

```
int i = 100;
printf("%d", i);
```

Good code???

```
int i = 100;
if (printf("%d", i) != 3)
   /* Error */
```

Is this overkill?

Internal Testing: Changing Code Temporarily

(3) Changing code temporarily

- Temporarily change code
  - To generate artificial boundary or stress tests

- Example: Array-based sorting program
  - Temporarily make array very small
  - Does the program handle overflow?

Leaving Testing Code Intact

(4) Leaving testing code intact

- Do not remove testing code when your code is finished
  - In industry, no code ever is “finished”!!!
- Leave tests in the code
- Maybe embed in calls of **`assert`**
  - Calls of **`assert`** can be disabled; described in precept

Internal Testing Summary

- Internal testing: Designing your program to test itself
- Internal testing techniques
  - (1) Checking invariants
  - (2) Checking function return values
  - (3) Changing code temporarily
  - (4) Leaving testing code intact

Beware: Do you see a conflict between internal testing and code clarity?

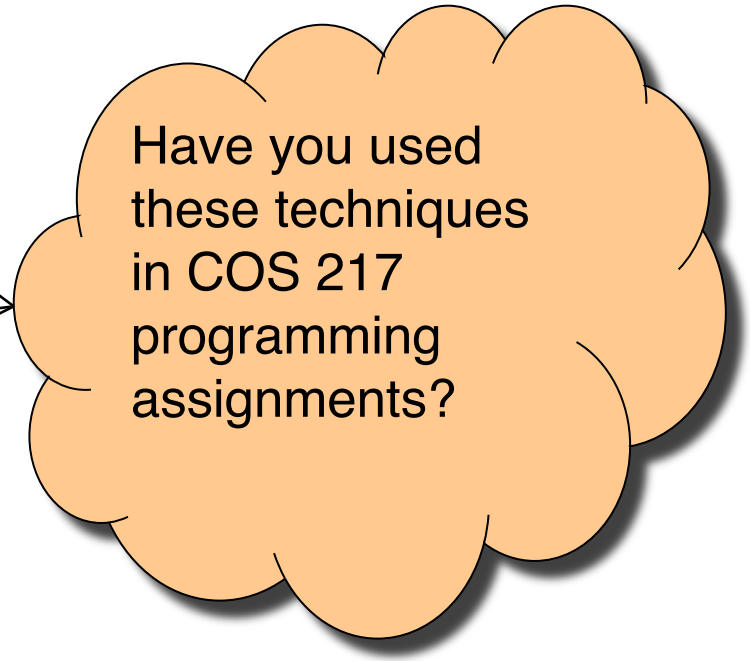
General Testing Strategies

- General testing strategies
  - Five strategies…

General Strategies: Automation

(1) Automation

- Create **scripts** and **data files** to test your **programs**
- Create **software clients** to test your **modules**
- Know what to expect
  - Generate output that is easy to recognize as right or wrong



- Automated testing can provide:
  - Much better coverage than manual testing
  - Bonus: Examples of typical use of your code

General Strategies: Testing Incrementally

(2) Testing incrementally

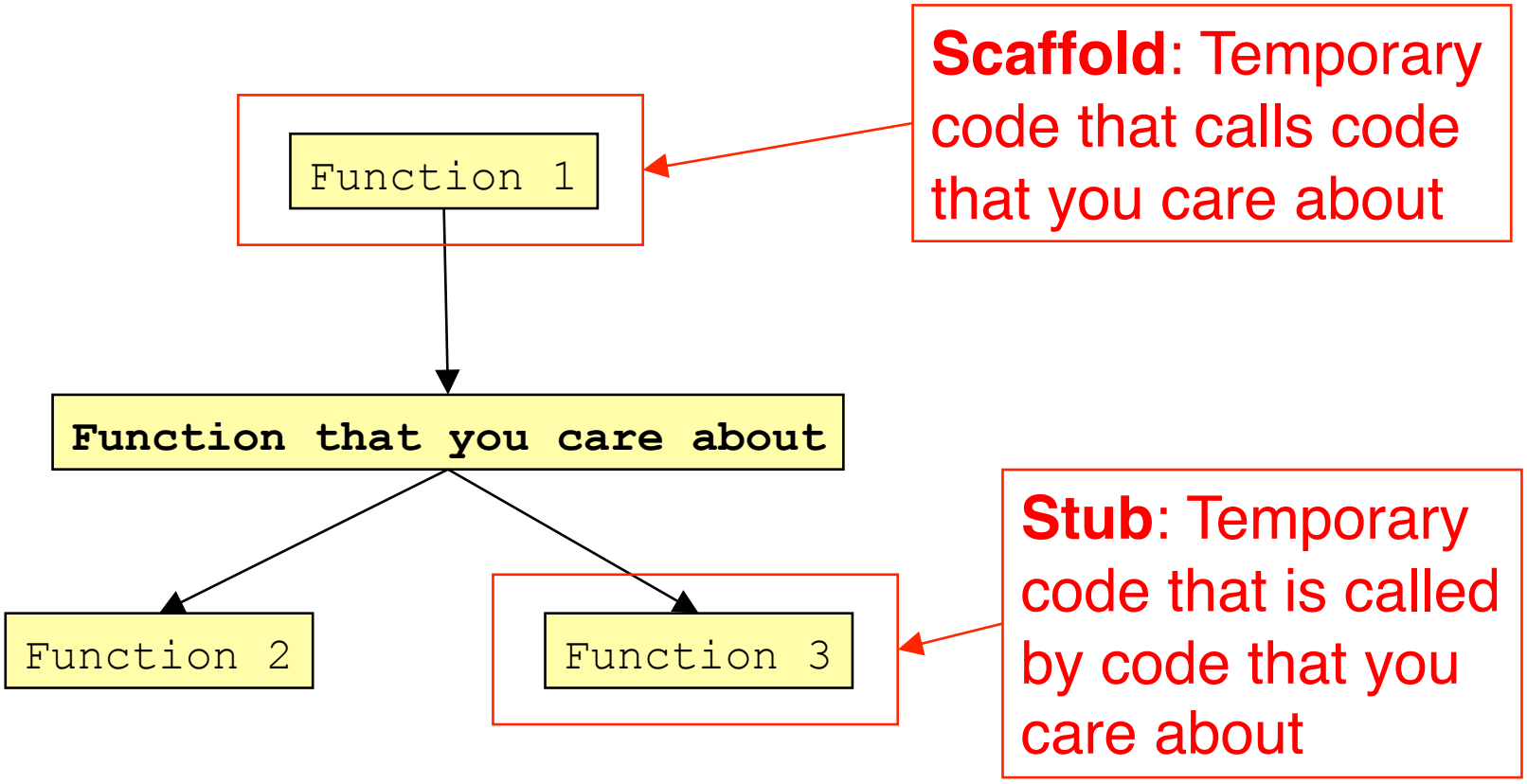
- Test as you write code
  - Add test cases as you create new code
  - Test individual modules, and then their interaction

- Do **regression testing**
  - After a bug fix, make sure program has not "regressed"
    - That is, make sure previously working code is not broken
  - Rerun **all** test cases
  - Note the value of automation!!!

Testing Incrementally (cont.)

(2) Testing incrementally (cont.)

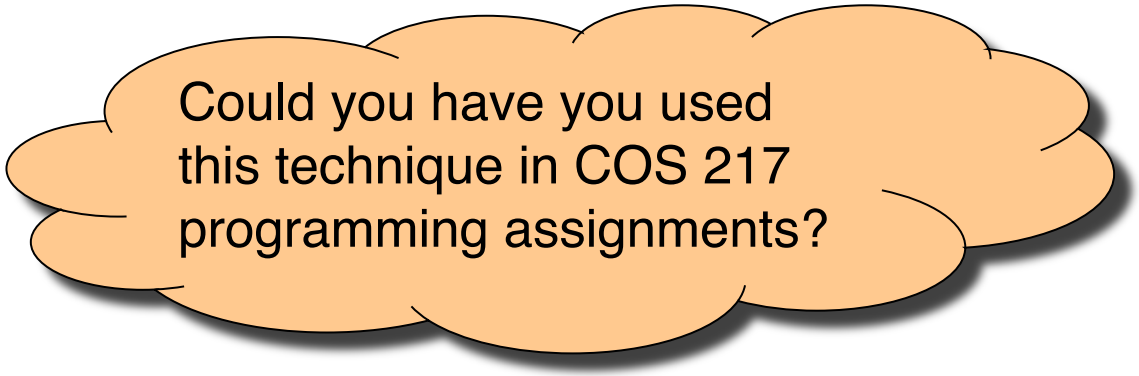
- Create **scaffolds** and **stubs** to test the code that you care about


Function 1
Function that you care about
Function 2
Function 3
Scaffold: Temporary code that calls code that you care about
Stub: Temporary code that is called by code that you care about

General Strategies: Comparing Implementations

(3) Comparing implementations

- Make sure independent implementations behave the same



General Strategies: Bug-Driven Testing

(4) Bug-driven testing

- Find a bug => create a test case that catches it
- Facilitates regression testing

General Strategies: Fault Injection

(5) Fault injection

- Intentionally (temporarily) inject bugs!!!
- Determine if testing finds them
- Test the testing!!!

General Strategies Summary

- General testing strategies
  - (1) Automation
  - (2) Testing incrementally
  - (3) Comparing implementations
  - (4) Bug-driven testing
  - (5) Fault injection

Who Tests What

- Programmers
  - **White-box** testing
  - Pro: Programmer knows all data paths
  - Con: Influenced by how code is designed/written
- Quality Assurance (QA) engineers
  - **Black-box** testing
  - Pro: No knowledge about the implementation
  - Con: Unlikely to test all logical paths
- Customers
  - **Field** testing
  - Pros: Unexpected ways of using the software; "debug" specs
  - Cons: Not enough cases; customers don't like "participating" in this process; malicious users exploit the bugs

Summary

- External testing taxonomy
  - Statement testing
  - Path testing
  - Boundary testing
  - Stress testing
- Internal testing techniques
  - Checking invariants
  - Checking function return values
  - Changing code temporarily
  - Leaving testing code intact
- General testing strategies
  - Automation
  - Testing incrementally
    - Regression testing
    - Scaffolds and stubs
  - Comparing implementations
  - Bug-driven testing
  - Fault injection
- Test the **code**, the **tests** – and the **specification**!

Goals of this Lecture

- Testing vs. debugging
  - Testing: What should I do to try to **break** my program?
  - Debugging: What should I do to try to **fix** my program

- Help you learn about:
  - Strategies and tools for debugging your code

- Why?
  - Debugging large programs can be difficult
  - Useful to know various debugging **strategies** and **tools**

Debugging Observations

- Most bugs are reproducible
  - Focus of inspection can be narrowed
  - Narrow by code path or by time
- Bugs are mismatches between expectation & execution
  - Can add more checks on expectations
  - Deviations detected early can prevent bugs
- Program flow can be watched
  - Printing and logging (especially high-volume)
  - Source-level debugging
- Not all bugs visibly manifested
  - But unmanifested bugs still exist
  - Classic cause of “the bug just went away”
  - Nothing ever just “goes away” in a deterministic world

Debugging Heuristics

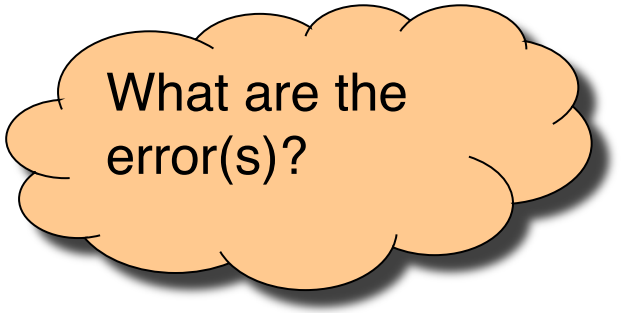
Debugging Heuristic	When Applicable
(1) Understand error messages	Build-time
(2) Think before writing	Run-time
(3) Look for familiar bugs	
(4) Divide and conquer	
(5) Add more internal tests	
(6) Display output	
(7) Use a debugger	
(8) Focus on recent changes	

Understand Error Messages

Debugging at **build-time** is easier than debugging at **run-time**, if and only if you…

(1) Understand the error messages!!!

```
#include <stdioo.h>
int main(void)
/* Print "hello, world" to stdout and
   return 0.
{
   printf("hello, world\n");
   return 0;
}
```



Understand Error Messages (cont.)

(1) Understand the error messages (cont.)

```
#include <stdioo.h>
int main(void)
/* Print "hello, world" to stdout and
   return 0.
{
   printf("hello, world\n");
   return 0;
}
```

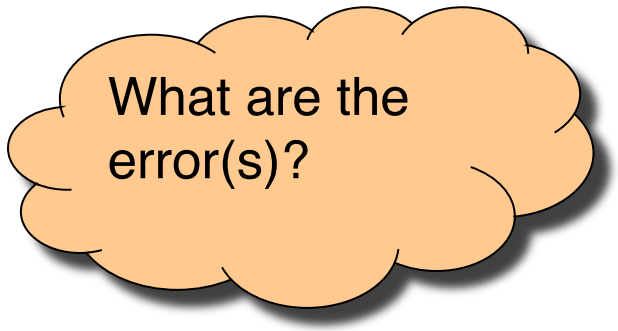
Which tool (preprocessor, compiler, or linker) reports the error(s)?

```
$ gcc217 hello.c -o hello
hello.c:1:20: stdioo.h: No such file or directory
hello.c:3:1: unterminated comment
hello.c:2: error: syntax error at end of input
```

Understand Error Messages (cont.)

(1) Understand the error messages (cont.)

```
#include <stdio.h>
int main(void)
/* Print "hello, world" to stdout and
   return 0. */
{
   printf("hello, world\n")
   retun 0;
}
```



Understand Error Messages (cont.)

(1) Understand the error messages (cont.)

```
#include <stdio.h>
int main(void)
/* Print "hello, world" to stdout and
   return 0. */
{
   printf("hello, world\n")
   retun 0;
}
```

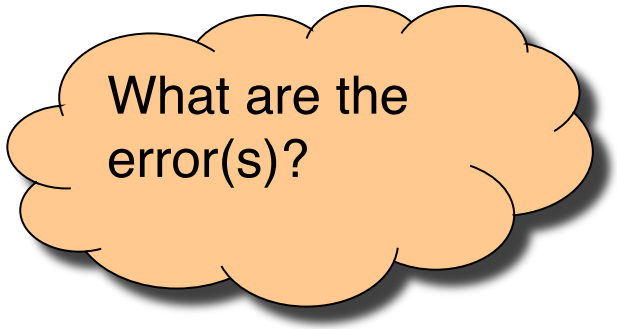
Which tool (preprocessor, compiler, or linker) reports the error(s)?

```
$ gcc217 hello.c -o hello
hello.c: In function `main':
hello.c:7: error: `retun' undeclared (first use in this
function)
hello.c:7: error: (Each undeclared identifier is reported
only once
hello.c:7: error: for each function it appears in.)
hello.c:7: error: syntax error before numeric constant
```

Understand Error Messages (cont.)

(1) Understand error messages (cont.)

```
#include <stdio.h>
int main(void)
/* Print "hello, world" to stdout and
   return 0. */
{
   prinf("hello, world\n")
   return 0;
}
```



Understand Error Messages (cont.)

(1) Understand error messages (cont.)

```
#include <stdio.h>
int main(void)
/* Print "hello, world" to stdout and
   return 0. */
{
   prinf("hello, world\n")
   return 0;
}
```

Which tool (preprocessor, compiler, or linker) reports the error(s)?

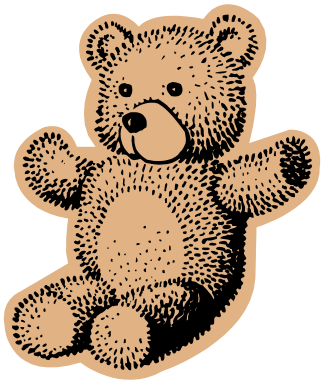
```
$ gcc217 hello.c -o hello
hello.c: In function `main':
hello.c:6: warning: implicit declaration of function
`prinf'
/tmp/cc43ebjk.o(.text+0x25): In function `main':
: undefined reference to `prinf'
collect2: ld returned 1 exit status
```

Think Before Writing

Inappropriate changes could make matters worse, so…

(2) Think before writing

- Draw pictures of the data structures
- Take a break
  - Sleep on it!
  - Start early so you can!!!
- Explain the code to:
  - Yourself
  - Someone else
  - A teddy bear!
  - A giant wookie!!!



Look for Familiar Bugs

(3) Look for familiar bugs

- Some of our favorites:

```
switch (i) {
   case 0:
      …
      break;
   case 1:
      …
   case 2:
      …
}
```

```
int i;
…
scanf("%d", i);
```

```
char c;
…
c = getchar();
```

```
while (c = getchar() != EOF)
   …
```

```
if (i = 5)
   …
```

```
if (5 < i < 10)
   …
```

```
if (i & j)
   …
```

What are the errors?

Look for Familiar Bugs

(3) Look in familiar places

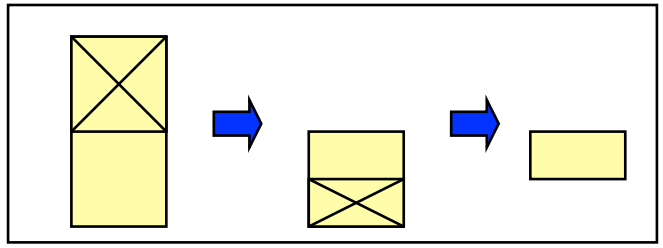
- Loop start & end conditions → "off by 1" errors
  - Most loop iterations run just fine
- Copy & pasted code
  - Brain sees main idea, not details
  - Details (like variable names) matter
- Check scoping, re-use of variables
  - Compiler complains about uninitialized use, not re-use

Divide and Conquer

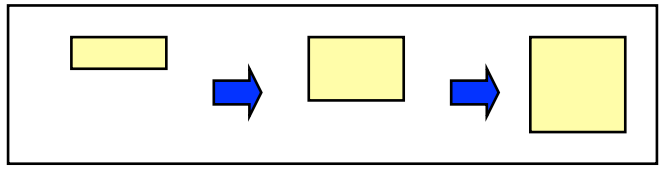
(4) Divide and conquer

- Incrementally find smallest/simplest **input** that illustrates the bug
- Example: Program fails on large input file *filex*

- Approach 1: **Remove** input
  - Start with *filex*
  - Incrementally remove lines of *filex* until bug disappears
    - Maybe in "binary search" fashion



- Approach 2: **Add** input
  - Start with small subset of *filex*
  - Incrementally add lines of *filex* until bug appears



Divide and Conquer (cont.)

(4) Divide and conquer (cont.)

- Incrementally find smallest **code subset** that illustrates the bug
- Example: Test client for your module fails

- Approach 1: **Remove** code
  - Start with test client
  - Incrementally remove lines of test client until bug disappears

- Approach 2: **Add** code
  - Start with minimal client
  - Incrementally add lines of test client until bug appears

Add More Internal Tests

(5) Add more internal tests

- Internal tests help **find** bugs (see “Testing” lecture)
- Internal test also can help **eliminate** bugs
  - Checking invariants and conservation properties can eliminate some functions from the bug hunt

Display Output

(6) Display output

- Print values of important variables at critical spots

- Poor:

```
printf("%d", keyvariable);
```

`stdout` is buffered; program may crash before output appears

- Maybe better:

```
printf("%d\n", keyvariable);
```

Printing `'\n'` flushes the `stdout` buffer, but not if stdout is redirected to a file

- Better:

```
printf("%d", keyvariable);
fflush(stdout);
```

Call `fflush()` to flush `stdout` buffer explicitly

Display Output (cont.)

(6) Display output (cont.)

- Maybe even better:

```
fprintf(stderr, "%d", keyvariable);
```

- Maybe better still:

```
FILE *fp = fopen("logfile", "w");
…
fprintf(fp, "%d", keyvariable);
fflush(fp);
```

Write debugging output to **`stderr`**; debugging output can be separated from normal output via redirection

Bonus: **`stderr`** is unbuffered

Write to a log file

Use a Debugger

(7) Use a debugger

- Alternative to displaying output
- Bonuses:
  - Debugger can load “core dumps”
    - Examine state of program when it terminated
  - Debugger can “attach” to a running program

The GDB Debugger

- **G**NU **Deb**ugger
  - Part of the GNU development environment
  - Integrated with Emacs editor
  - Allows user to:
    - Run program
    - Set breakpoints
    - Step through code one line at a time
    - Examine values of variables during run
    - Etc.

Using GDB: Example Program

- An example program

  File testintmath.c:

Euclid's algorithm; Don't be concerned with details

```
#include <stdio.h>

int gcd(int i, int j) {
   int temp;
   while (j != 0) {
      temp = i % j;
      i = j;
      j = temp;
   }
   return i;
}

int lcm(int i, int j) {
   return (i / gcd(i, j)) * j;
}
…
```

```
…
int main(void) {
   int iGcd;
   int iLcm;
   iGcd = gcd(8, 12);
   iLcm = lcm(8, 12);
   printf("%d %d\n", iGcd, iLcm);
   return 0;
}
```

The program is correct

But let's pretend it has a runtime error in **`gcd()`**…

Using GDB: Compile, Run, Break

(a) Build with –g

```
gcc217 -g testintmath.c -o testintmath
```

- Adds extra information to executable file that GDB uses

(b) Run Emacs, with no arguments

```
emacs
```

(c) Run GDB on executable file from within Emacs

```
<Esc key> x gdb <Enter key> testintmath <Enter key>
```

(d) Set breakpoints, as desired

```
break main
```

- GDB sets a breakpoint at the first executable line of main()

```
break gcd
```

- GDB sets a breakpoint at the first executable line of gcd()

Using GDB: Stepping Through Code

- Typical steps for using GDB (cont.):

  (e) Run the program

  `run`
  - GDB stops at the breakpoint in main()
  - Emacs opens window showing source code
  - Emacs highlights line that is to be executed next

  `continue`
  - GDB stops at the breakpoint in gcd()
  - Emacs highlights line that is to be executed next

  (f) Step through the program, as desired

  `step` (repeatedly)
  - GDB executes the next line (repeatedly)

- Note: When next line is a call of one of your functions:
  - `step` command *steps into* the function
  - `next` command *steps over* the function, that is, executes the next line without stepping into the function

Using GDB: Examining Variables

(g) Examine variables, as desired

```
print i
print j
print temp
```

- GDB prints the value of each variable

(h) Examine the function call stack, if desired

```
where
```

- GBB prints the function call stack
- Useful for diagnosing crash in large program

(i) Exit gdb

```
quit
```

(j) Exit Emacs

```
<Ctrl-x key> <Ctrl-c key>
```

Using GDB: Sophisticated I/O

- GDB can do much more:
  - Handle command-line arguments

    ```
    run arg1 arg2
    ```
  - Handle redirection of stdin, stdout, stderr

    ```
    run < somefile > someotherfile
    ```
  - Print values of expressions
  - Break conditionally
  - Etc.
- See *Programming with GNU Software* (Loukides and Oram) Chapter 6

Focus on Recent Changes

(8) Focus on recent changes

- Corollary: Debug now, not later

Difficult:

(1) Write entire program
(2) Test entire program
(3) Debug entire program

Easier:

(1) Write a little
(2) Test a little
(3) Debug a little
(4) Write a little
(5) Test a little
(6) Debug a little
…

Focus on Recent Changes (cont.)

(8) Focus on recent change (cont.)

- Corollary: Maintain old versions

Difficult:

(1) Change code
(2) Note bug
(3) Try to remember what changed since last working version!!!

Easier:

(1) Backup working version
(2) Change code
(3) Note bug
(4) Compare code with working version to determine what changed

Maintaining Previous Versions

- To maintain old versions
  - Approach 1: Manually copy project directory

```
…
$ mkdir myproject
$ cd myproject

   Create project files here.

$ cd ..
$ cp -r myproject myprojectDateTime
$ cd myproject

   Continue creating project files here.
…
```

  - Approach 1.5: use snapshot support in filesystem
  - Approach 2: Use RCS…

RCS

Revision Control System

- A simple personal version control system
- Provided with many Linux distributions
  - Available on hats
- Allows developer to:
  - Create a **source code repository**
  - **Check** source code files **into** repository
    - RCS saves old versions
  - **Check** source code files **out of** repository

- Appropriate for **one-developer** projects
- **Not** appropriate for **multi-developer** projects
  - Use **CVS** or **Subversion** instead

Using RCS: Getting Started

(a) Create project directory, as usual

```
mkdir helloproj
cd helloproj
```

(b) Create RCS directory in project directory

```
mkdir RCS
```

- RCS will store its repository in that directory

(c) Create source code files in project directory

```
emacs hello.c …
```

(d) Check in

```
ci hello.c
```

- Adds file to RCS repository
- Deletes local copy (don't panic!)
- Can provide description of file (1st time)
- Can provide log message, typically describing changes

Using RCS: Check Out and In

(e) Check out most recent version for reading

```
co hello.c
```

- Copies file from repository to project directory
- File in project directory has read-only permissions

(f) Check out most recent version for reading/writing

```
co -l hello.c
```

- Copies file from repository to project directory
- File in project directory has read/write permissions

(g) List versions in repository

```
rlog hello.c
```

- Shows versions of file, by number (1.1, 1.2, etc.), with descriptions

(h) Check out a specified version

```
co -l -rversionnumber hello.c
```

Using RCS: Fancy Stuff

- RCS can do much more:
  - Merge versions of files
  - Maintain distinct development branches
  - Place descriptions in code as comments
  - Assign symbolic names to versions
  - Etc.
- See *Programming with GNU Software* (Loukides and Oram) Chapter 8
- Recommendation: Use RCS
  - `ci` and `co` can become automatic!

Summary

Debugging Heuristic	When Applicable
(1) Understand error messages	Build-time
(2) Think before writing	Run-time
(3) Look for familiar bugs	
(4) Divide and conquer	
(5) Add more internal tests	
(6) Display output	
(7) Use a debugger *	
(8) Focus on recent changes **	

* Use GDB
** Use RCS

See Appendix slides on debugging memory management code…

Appendix: Debugging Mem Mgmt

- Some debugging techniques are specific to **dynamic memory management**
  - That is, to memory managed by `malloc()`, `calloc()`, `realloc()`, and `free()`
- Soon will be pertinent in the course
- For future reference…

Appendix: Debugging Mem Mgmt (cont.)

(9) Look for familiar dynamic memory management bugs

- Some of our favorites:

```
int *p; /* value of p undefined */
…
*p = somevalue;
```

Dangling pointer

```
int *p; /* value of p undefined */
…
fgets(p, 1024, stdin);
```

Dangling pointer

```
int *p;
…
p = (int*)malloc(sizeof(int));
…
free(p);
…
*p = 5;
```

Dangling pointer

Appendix: Debugging Mem Mgmt (cont.)

(9) Look for familiar dynamic memory management bugs (cont.)

- Some of our favorites (cont.):

```
int *p;
…
p = (int*)malloc(sizeof(int));
…
p = (int*)malloc(sizeof(int));
…
```

Memory leak
alias
Garbage creation

Detection: Valgrind, etc.

```
int *p;
…
p = (int*)malloc(sizeof(int));
…
free(p);
…
free(p);
```

Multiple free

*Detection: man malloc, MALLOC_CHECK_*

Appendix: Debugging Mem Mgmt (cont.)

(9) Look for familiar dynamic memory management bugs (cont.)

- Some of our favorites (cont.):

```
char *s1 = "Hello";
char *s2;
s2 = (char*)malloc(strlen(s1));
strcpy(s2, s1);
```

Allocating too few bytes

```
char *s1 = "Hello";
char *s2;
s2 = (char*)malloc(sizeof(s1));
strcpy(s2, s1);
```

Allocating too few bytes

```
double *p;
p = (double*)malloc(sizeof(double*));
```

Allocating too few bytes

Appendix: Debugging Mem Mgmt (cont.)

(10) Segmentation fault? Make it happen within gdb, and then issue the gdb **`where`** command. The output will lead you to the line that caused the fault. (But that line may not be where the error resides.)

(11) Manually inspect each call of malloc(), calloc(), and realloc() in your code, making sure that it allocates enough memory.

(12) Temporarily hardcode each call of malloc(), calloc(), and realloc() such that it requests a large number of bytes. If the error disappears, then you'll know that at least one of your calls is requesting too few bytes.

Appendix: Debugging Mem Mgmt (cont.)

(13) Temporarily comment-out each call of free() in your code. If the error disappears, then you'll know that you're freeing memory too soon, or freeing memory that already has been freed, or freeing memory that should not be freed, etc.

(14) Use the Meminfo tool. Programs built with gcc217m are much more sensitive to dynamic memory management errors than are programs built with gcc217. So the error might manifest itself earlier, and thereby might be easier to diagnose.

Programming and Program Style

Jennifer Rexford

The material for this lecture is drawn, in part, from
The Practice of Programming (Kernighan & Pike) Chapter 1

Goals of This Lecture

- Help you learn about:
  - Good **programming** (verb) style
  - Good **program** (noun) style

- Why?
  - A well-styled program is **easier to maintain**
  - A well-styled program is **more likely to be correct**

Lecture Overview

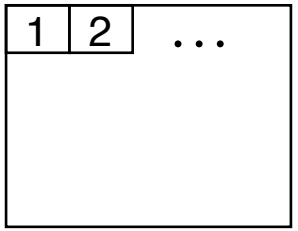
- **Programming style**: creating a good program
  - Top-down design
  - Successive refinement
  - Example: left and right justifying text
- **Program style**: qualities of a good program
  - Well structured
  - Uses common idioms
  - Uses descriptive names
  - Contains proper comments
  - Modular

Part 1: Programming Style

Bottom-Up Design is Bad

- Bottom-up design ☹
  - Design one part in detail
  - Design another part in detail
  - Repeat until finished

- Bottom-up design in **painting**
  - Paint upper left part, paint next part, …
  - *Unlikely to produce a good painting*

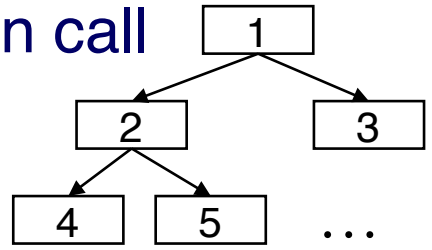


- Bottom-up design in **programming**
  - Write first part, write next part of program, ….
  - *Unlikely to produce a good program*

Top-Down Design is Good

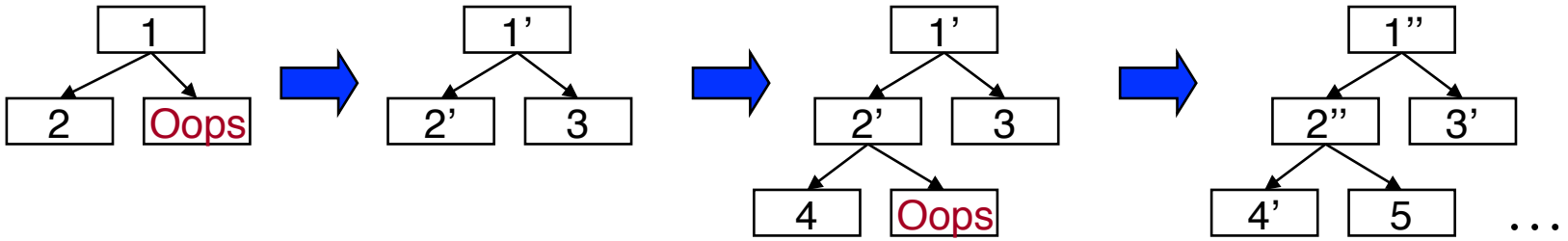
- Top-down design ☺
  - Design entire product with minimal detail
  - Successively refine until finished
- Top-down design in **painting**
  - Sketch the entire painting with minimal detail
  - Successively refine the entire painting

- Top-down design in **programming**
  - Define main() function in pseudocode
  - Refine each statement with code or function call
  - Recurse in (mostly) breadth-first order
  - Bonus: Product is naturally modular



Top-Down Design in Reality

- Top-down design in programming **in reality**
  - Define main() function in pseudocode
  - Refine each pseudocode statement
    - Oops! Details reveal design error, so…
    - Backtrack to refine existing (pseudo)code
  - Recurse in (mostly) breadth-first order
    - Until all functions are defined



Illustrative Example

- Illustrate good programming style
  - Especially function-level modularity and top-down design
- Illustrate going from problem statement to code
  - Review and illustrate C constructs
- Example: text formatting
  - Derived from King Section 15.3

Example: Text Formatting

- Input: ASCII text
  - Text with arbitrary spaces & newlines
- Output: the same text, left and right justified
  - Fit as many words as possible on each 50-character line
  - Add even spacing between words to right justify the text
  - No need to right justify the very last line
- Simplifying assumptions
  - Word ends at white space or end-of-file
  - No word is longer than 20 characters

Example Input and Output

INPUT

```
Tune   every   heart    and    every      voice.
Bid every      bank withdrawal.
Let's all    with our    accounts  rejoice.
In funding  Old  Nassau.
In funding Old Nassau we spend more money every year.
Our banks      shall give, while      we     shall    live.
We're funding                    Old Nassau.
```

OUTPUT

```
Tune every heart and every voice. Bid  every  bank
withdrawal. Let's all with our  accounts  rejoice.
In funding Old Nassau. In funding  Old  Nassau  we
spend more money every year. Our banks shall give,
while we shall live. We're funding Old Nassau.
```

Thinking About the Problem

- I need a notion of "word"
  - Sequence of characters with no white space
  - All characters in a word must be printed on the same line
- I need to be able to read and print words
  - Read characters from **`stdin`** till white space or **`EOF`**
  - Print characters to **`stdout`** followed by space(s) or newline
- I need to deal with poorly-formatted input
  - I need to remove extra white space in input
- Unfortunately, I can't print the words as they are read
  - I don't know # of spaces needed till I read the future words
  - Need to buffer the words until I can safely print an entire line
- But, how much space should I add between words?
  - Need at least one space between adjacent words on a line
  - Can add extra spaces evenly to fill up an entire line

Writing the Program

- Key constructs
  - Word
  - Line
- Next steps
  - Write pseudocode for **`main()`**
  - Successively refine
- Caveats concerning the following presentation
  - Function comments and some blank lines are omitted because of space constraints
    - Don't do that in practice!!!
  - Design sequence is idealized
    - In reality, much backtracking would occur

The Top Level

- First, let's sketch **`main()`** …

```
int main(void) {
   <Clear line>
   for (;;) {
      <Read a word>
      if (<No more words>) {
         <Print line with no justification>
         return 0;
      }
      if (<Word doesn't fit on this line>) {
         <Print line with justification>
         <Clear line>
      }
      <Add word to line>
   }
   return 0;
}
```

Reading a Word

```
…
enum {MAX_WORD_LEN = 20};
int main(void) {
   char word[MAX_WORD_LEN + 1];
   int wordLen;
   <Clear line>
   for (;;) {
      wordLen = ReadWord(word);
      if (<No more words>) {
         <Print line with no justification>
         return 0;
      }
      if (<Word doesn't fit on this line>) {
         <Print line with justification>
         <Clear line>
      }
      <Add word to line>
   }
   return 0;
}
```

- Now let's successively refine. What does <Read a word> mean? The job seems complicated enough that it should be delegated to a distinct function…

```
int ReadWord(char *word) {
   <Skip over whitespace>
   <Store chars up to MAX_WORD_LEN in word>
   <Return length of word>
}
```

Reading a Word (cont.)

- **`ReadWord()`** seems easy enough to design. So let's flesh it out…

```
int ReadWord(char *word) {
   int ch, pos = 0;

   /* Skip over white space. */
   ch = getchar();
   while ((ch != EOF) && isspace(ch))
      ch = getchar();

   /* Store chars up to MAX_WORD_LEN in word. */
   while ((ch != EOF) && (! isspace(ch))) {
      if (pos < MAX_WORD_LEN) {
         word[pos] = (char)ch;
         pos++;
      }
      ch = getchar();
   }
   word[pos] = '\0';

   /* Return length of word. */
   return pos;
}
```

Saving a Word

```
…
enum {MAX_WORD_LEN = 20};
enum {MAX_LINE_LEN = 50};
int main(void) {
   char word[MAX_WORD_LEN + 1];
   int wordLen;
   char line[MAX_LINE_LEN + 1];
   int lineLen = 0;
   <Clear line>
   for (;;) {
      wordLen = ReadWord(word);
      if (<No more words>) {
         <Print
         return
      }
      if (<Word
         <Print
         <Clear
      }
      AddWord(word, line, &lineLen);
   }
   return 0;
}
```

```
void AddWord(const char *word, char *line, int *lineLen) {
   <if line already contains some words, append a space>
   strcat(line, word);
   (*lineLen) += strlen(word);
}
```

- Now, back to **`main()`**. What does <Add word to line> mean? The job seems complicated enough to demand a distinct function…

Saving a Word (cont.)

- **`AddWord()`** is almost complete already, so let's get that out of the way...

```
void AddWord(const char *word, char *line, int *lineLen) {

   /* If line already contains some words, append a space. */
   if (*lineLen > 0) {
      line[*lineLen] = ' ';
      line[*lineLen + 1] = '\0';
      (*lineLen)++;
   }

   strcat(line, word);
   (*lineLen) += strlen(word);
}
```

Printing the Last Line

```
…
int main(void) {
   char word[MAX_WORD_LEN + 1];
   int wordLen;
   char line[MAX_LINE_LEN + 1];
   int lineLen = 0;
   <Clear line buffer>
   for (;;) {
      wordLen = ReadWord(word);

      /* If no more words, print line
         with no justification. */
      if ((wordLen == 0) && (lineLen > 0)) {
         puts(line);
         return 0;
      }
      if (<Word doesn't fit on this line>) {
         <Print line with justification>
         <Clear line buffer>
      }
      AddWord(word, line, &lineLen);
   }
   return 0;
}
```

- Again, back to **`main()`**. What do <No more words> and <Print line with no justification> mean? Those jobs seem easy enough that we need not define additional functions…

Deciding When to Print

```
…
int main(void) {
   char word[MAX_WORD_LEN + 1];
   int wordLen;
   char line[MAX_LINE_LEN + 1];
   int lineLen = 0;
   <Clear line buffer>
   for (;;) {
      wordLen = ReadWord(word);

      /* If no more words, print line
         with no justification. */
      if ((wordLen == 0) && (lineLen > 0)) {
         puts(line);
         return 0;
      }
      /* If word doesn't fit on this line, then… */
      if ((wordLen + 1 + lineLen) > MAX_LINE_LEN) {
         <Print line with justification>
         <Clear line buffer>
      }
      AddWord(word, line, &lineLen);
   }
   return 0;
}
```

- What does <Word doesn't fit on this line> mean? That's somewhat tricky, but involves little code…

Printing with Justification

- Now, to the heart of the program. What does <Print line with justification> mean? Certainly that job demands a distinct function. Moreover, it's clear that the function must know how many words are in the given line. So let's change **`main()`** accordingly…

```
…
int main(void) {
   …
   int numWords = 0;
   <Clear line>
   for (;;) {
      …
      /* If word doesn't fit on this line, then… */
      if ((wordLen + 1 + lineLen) > MAX_LINE_LEN) {
         WriteLine(line, lineLen, numWords);
         <Clear line>
      }

      AddWord(word, line, &lineLen);
      numWords++;
   }
   return 0;
}
```

Printing with Justification (cont.)

- And write pseudocode for **`WriteLine()`**…

```
void WriteLine(const char *line, int lineLen, int numWords) {

  <Compute number of excess spaces for line>

  for (i = 0; i < lineLen; i++) {
    if (<line[i] is not a space>)
      <Print the character>
    else {
      <Compute additional spaces to insert>

      <Print a space, plus additional spaces>

      <Decrease extra spaces and word count>
    }
  }
}
```

Printing with Justification (cont.)

```
void WriteLine(const char *line, int lineLen, int numWords)
{
   int extraSpaces, spacesToInsert, i, j;

   /* Compute number of excess spaces for line. */
   extraSpaces = MAX_LINE_LEN - lineLen;

   for (i = 0; i < lineLen; i++) {
      if (line[i] != ' ')
         putchar(line[i]);
      else {
         /* Compute additional spaces to insert. */
         spacesToInsert = extraSpaces / (numWords - 1);

         /* Print a space, plus additional spaces. */
         for (j = 1; j <= spacesToInsert + 1; j++)
            putchar(' ');

         /* Decrease extra spaces and word count. */
         extraSpaces -= spacesToInsert;
         numWords--;
      }
   }
   putchar('\n');
}
```

- Let's go ahead and complete **`WriteLine()`**

The number of gaps

Example:
If extraSpaces is 10 and numWords is 5, then gaps will contain 2, 2, 3, and 3 extra spaces respectively

Clearing the Line

- One step remains. What does <Clear line> mean? It's an easy job, but it's done in two places. So we probably should delegate the work to a distinct function, and call the function in the two places…

```
…
int main(void) {
   …
   int numWords = 0;
   ClearLine(line, &lineLen, &numWords);
   for (;;) {
      …
      /* If word doesn't fit on this line, then… */
      if ((wordLen + 1 + lineLen) > MAX_LINE_LEN) {
         WriteLine(line, lineLen, numWords);
         ClearLine(line, &lineLen, &numWords);
      }

      addWord(wo
      numWords++
   }
   return 0;
}
```

```
void ClearLine(char *line, int *lineLen, int *numWords) {
   line[0] = '\0';
   *lineLen = 0;
   *numWords = 0;
}
```

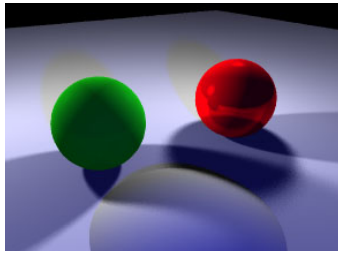
Modularity: Summary of Example

- To the user of the program
  - Input: Text in messy format
  - Output: Same text left and right justified
- Between parts of the program
  - Word-handling functions
  - Line-handling functions
  - `main()` function
- The many benefits of modularity
  - Reading the code: In small, separable pieces
  - Testing the code: Test each function separately
  - Speeding up the code: Focus only on the slow parts
  - Extending the code: Change only the relevant parts

Part 2: Program Style

Program Style

- Who reads your code?
  - The compiler
  - Other programmers



```
typedef struct{double x,y,z}vec;vec U,black,amb={.02,.02,.02};struct sphere{ vec
cen,color;double rad,kd,ks,kt,kl,ir}*s,*best,sph[]={0.,6.,.5,1.,1.,1.,.9, .
05,.2,.85,0.,1.7,-1.,8.,-.5,1.,.5,.2,1.,.7,.3,0.,.05,1.2,1.,8.,-.5,.1,.8,.8,
1.,.3,.7,0.,0.,1.2,3.,-6.,15.,1.,.8,1.,7.,0.,0.,0.,.6,1.5,-3.,-3.,12.,.8,1.,
1.,5.,0.,0.,0.,.5,1.5,};yx;double u,b,tmin,sqrt(),tan();double vdot(A,B)vec A ,B;{return
A.x*B.x+A.y*B.y+A.z*B.z;}vec vcomb(a,A,B)double a;vec A,B;{B.x+=a* A.x;B.y+=a*A.y;B.z
+=a*A.z;return B;}vec vunit(A)vec A;{return vcomb(1./sqrt( vdot(A,A)),A,black);}struct
sphere*intersect(P,D)vec P,D;{best=0;tmin=1e30;s= sph+5;while(s--sph)b=vdot(D,U=vcomb(-1.,P,s-
cen)),u=b*b-vdot(U,U)+s-rad*s -rad,u=u0?sqrt(u):1e31,u=b-u1e-7?b-u:b+u,tmin=u=1e-7&&u<tmin?
best=s,u: tmin;return best;}vec trace(level,P,D)vec P,D;{double d,eta,e;vec N,color; struct
sphere*s,*l;if(!level--)return black;if(s=intersect(P,D));else return amb;color=amb;eta=s-ir;d=
-vdot(D,N=vunit(vcomb(-1.,P=vcomb(tmin,D,P),s-cen )));if(d<0)N=vcomb(-1.,N,black),eta=1/eta,d=
-d;l=sph+5;while(l--sph)if((e=l -kl*vdot(N,U=vunit(vcomb(-1.,P,l-cen))))0&&intersect(P,U)==l)
color=vcomb(e ,l-color,color);U=s-color;color.x*=U.x;color.y*=U.y;color.z*=U.z;e=1-eta* eta*(1-
d*d);return vcomb(s-kt,e0?trace(level,P,vcomb(eta,D,vcomb(eta*d-sqrt (e),N,black))):black,vcomb
(s-ks,trace(level,P,vcomb(2*d,N,D)),vcomb(s-kd, color,vcomb(s-kl,U,black))));}main(){printf("%d
%d\n",32,32);while(yx<32*32) U.x=yx%32-32/2,U.z=32/2-yx++/32,U.y=32/2/tan
(25/114.5915590261),U=vcomb(255., trace(3,black,vunit(U)),black),printf("%.0f %.0f %.0f\n",U);}
```

This is a working ray tracer! (courtesy of Paul Heckbert)

Program Style

- Why does program style matter?
  - Bugs often caused by programmer's misunderstanding
    - What does this variable do?
    - How is this function called?
  - Good code = human readable code
- How can code become easier for humans to read?
  - Convey program structure
  - Use common idioms
  - Choose descriptive names
  - Compose proper comments
  - Use modularity

Structure: Spacing

- Use readable/consistent spacing
  - Example: Assign each array element a[j] to the value j.
  - Bad code

```
for (j=0;j<100;j++) a[j]=j;
```

  - Good code

```
for (j = 0; j < 100; j++)
   a[j] = j;
```

  - Often can rely on auto-indenting feature in editor

Structure: Indentation (cont.)

- Use readable/consistent/correct indentation
  - Example: Checking for leap year (does Feb 29 exist?)

```
legal = TRUE;
if (month == FEB) {
   if (year % 4 == 0)
      if (day > 29)
         legal = FALSE;
   else
      if (day > 28)
         legal = FALSE;
}
```

Does this code work?

```
legal = TRUE;
if (month == FEB) {
   if (year % 4 == 0) {
      if (day > 29)
         legal = FALSE;
   }
   else {
      if (day > 28)
         legal = FALSE;
   }
}
```

Does this code work?

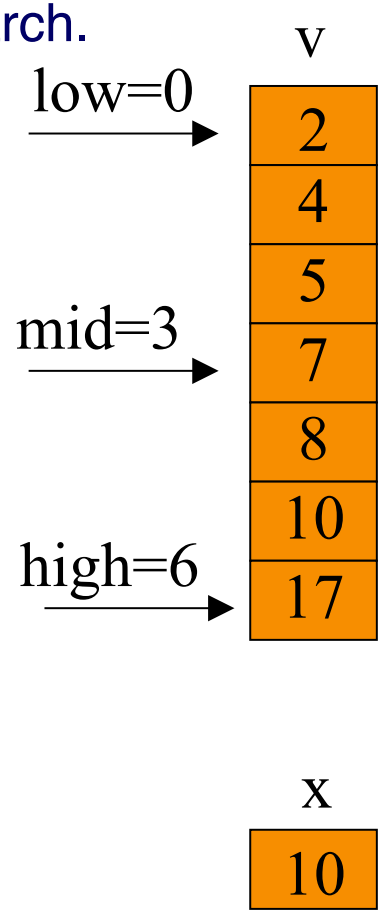
Structure: Indentation (cont.)

- Use “else-if” for multi-way decision structures
  - Example: Comparison step in a binary search.
  - Bad code

```
if (x < v[mid])
    high = mid - 1;
else
    if (x > v[mid])
        low = mid + 1;
    else
        return mid;
```

  - Good code

```
if (x < v[mid])
    high = mid - 1;
else if (x > v[mid])
    low = mid + 1;
else
    return mid;
```



Structure: “Paragraphs”

- Use blank lines to divide the code into key parts

```
#include <stdio.h>
#include <stdlib.h>

int main(void)

/* Read a circle's radius from stdin, and compute and write its
   diameter and circumference to stdout.  Return 0 if successful. */

{
   const double PI = 3.14159;
   int radius;
   int diam;
   double circum;

   printf("Enter the circle's radius:\n");
   if (scanf("%d", &radius) != 1)
   {
      fprintf(stderr, "Error: Not a number\n");
      exit(EXIT_FAILURE);  /* or:  return EXIT_FAILURE; */
   }
...
```

Structure: “Paragraphs”

- Use blank lines to divide the code into key parts

```
   diam = 2 * radius;
   circum = PI * (double)diam;

   printf("A circle with radius %d has diameter %d\n",
      radius, diam);
   printf("and circumference %f.\n", circum);

   return 0;
}
```

Structure: Expressions

- Use natural form of expressions
  - Example: Check if integer `n` satisfies `j < n < k`
  - Bad code

```
if (!(n >= k) && !(n <= j))
```

  - Good code

```
if ((j < n) && (n < k))
```

  - Conditions should read as you'd say them aloud
    - Not "Conditions shouldn't read as you'd never say them aloud"!

Structure: Expressions (cont.)

- Parenthesize to resolve ambiguity
  - Example: Check if integer `n` satisfies `j < n < k`
  - Bad code

```
if (j < n && n < k)
```

Does this code work?

  - Good code

```
if ((j < n) && (n < k))
```

Structure: Expressions (cont.)

- Parenthesize to resolve ambiguity (cont.)
  - Example: read and print character until end-of-file
  - Bad code

```
while (c = getchar() != EOF)
   putchar(c);
```

Does this code work?

  - Good code

```
while ((c = getchar()) != EOF)
   putchar(c);
```

Structure: Expressions (cont.)

- Break up complex expressions
  - Example: Identify chars corresponding to months of year
  - Bad code

```
if ((c == 'J') || (c == 'F') || (c ==
'M') || (c == 'A') || (c == 'S') || (c
== 'O') || (c == 'N') || (c == 'D'))
```

  - Good code

```
if ((c == 'J') || (c == 'F') ||
    (c == 'M') || (c == 'A') ||
    (c == 'S') || (c == 'O') ||
    (c == 'N') || (c == 'D'))
```

  - Lining up the parallel structures is helpful, too!

C Idioms

- Use C idioms
  - Example: Set each array element to 1.0.
  - Bad code (or, perhaps just "so-so" code)

```
i = 0;
while (i <= n-1)
   array[i++] = 1.0;
```

  - Good code

```
for (i = 0; i < n; i++)
   array[i] = 1.0;
```

  - We'll see many C idioms throughout the course
  - Don't feel obliged to use C idioms that decrease clarity

Naming

- Use descriptive names for globals and functions
  - E.g., **`display, CONTROL, CAPACITY`**
- Use concise names for local variables
  - E.g., **`i`** (not **`arrayIndex`**) for loop variable
- Use case judiciously
  - E.g., Buffer_insert (Module_function)
    CAPACITY (constant)
    buf (local variable)
- Use a consistent style for compound names
  - E.g., **`frontsize`**, **`frontSize`**, **`front_size`**
- Use active names for functions
  - E.g., **`getchar()`**, **`putchar()`**, **`Check_octal()`**, etc.

Comments

- Master the language and its idioms
  - Let the code speak for itself
  - And then…
- Compose comments that add new information

```
i++;    /* add one to i */
```

- Comment sections ("paragraphs") of code, not lines of code
  - E.g., "Sort array in ascending order"
- Comment global data
  - Global variables, structure type definitions, field definitions, etc.
- Compose comments that agree with the code!!!
  - And change as the code itself changes. ☺

Comments (cont.)

- Comment sections (“paragraphs”) of code, not lines of code

```
#include <stdio.h>
#include <stdlib.h>

int main(void)

/* Read a circle's radius from stdin, and compute and write its
   diameter and circumference to stdout.  Return 0 if successful. */

{
   const double PI = 3.14159;
   int radius;
   int diam;
   double circum;

   /* Read the circle's radius. */
   printf("Enter the circle's radius:\n");
   if (scanf("%d", &radius) != 1)
   {
      fprintf(stderr, "Error: Not a number\n");
      exit(EXIT_FAILURE);  /* or:  return EXIT_FAILURE; */
   }
...
```

Comments (cont.)

```
   /* Compute the diameter and circumference. */
   diam = 2 * radius;
   circum = PI * (double)diam;

   /* Print the results. */
   printf("A circle with radius %d has diameter %d\n",
      radius, diam);
   printf("and circumference %f.\n", circum);

   return 0;
}
```

Function Comments

- Describe **what a caller needs to know** to call the function properly
  - Describe **what the function does**, not **how it works**
  - Code itself should clearly reveal how it works…
  - If not, compose "paragraph" comments within definition
- Describe **input**
  - Parameters, files read, global variables used
- Describe **output**
  - Return value, parameters, files written, global variables affected
- Refer to parameters **by name**

Function Comments (cont.)

- Bad function comment

```
/* decomment.c */

int main(void) {

/* Read a character.  Based upon the character and
   the current DFA state, call the appropriate
   state-handling function.  Repeat until
   end-of-file. */

   …
}
```

- Describes **how the function works**

Function Comments (cont.)

- Good function comment

```
/* decomment.c */

int main(void) {

/* Read a C program from stdin.  Write it to
   stdout with each comment replaced by a single
   space.  Preserve line numbers.  Return 0 if
   successful, EXIT_FAILURE if not. */

   …
}
```

- Describes **what the function does**

Modularity

- Big programs are harder to write than small ones
  - "A dog house can be built without any particular design, using whatever materials are at hand. A house for humans, on the other hand, is too complex to just throw together." – K. N. King
- Abstraction is the key to managing complexity
  - Abstraction allows programmer to know *what* something does without knowing *how*
- Examples of function-level abstraction
  - Function to sort an array of integers
  - Character I/O functions such as **`getchar()`** and **`putchar()`**
  - Mathematical functions such as **`lcm()`** and **`gcd()`**
- Examples of file-level abstraction
  - (Described in a later lecture)

Summary

- Programming style
  - Think about the problem
  - Use top-down design and successive refinement
  - But know that backtracking inevitably will occur
- Program style
  - Convey structure (spacing, indentation, parentheses)
  - Use common C idioms
  - Choose descriptive names for variables, functions
  - Compose proper comments, especially for functions
  - Divide code into modules (functions and files)

Appendix: The "justify" Program

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

enum {MAX_WORD_LEN = 20};
enum {MAX_LINE_LEN = 50};
```

Continued on next slide

Appendix: The "justify" Program

```
int ReadWord(char *word) {

/* Read a word from stdin.  Assign it to word.  Return the length
   of the word, or 0 if no word could be read. */

   int ch, pos = 0;

   /* Skip over white space. */
   ch = getchar();
   while ((ch != EOF) && isspace(ch))
      ch = getchar();

   /* Store chars up to MAX_WORD_LEN in word. */
   while ((ch != EOF) && (! isspace(ch))) {
      if (pos < MAX_WORD_LEN) {
         word[pos] = (char)ch;
         pos++;
      }
      ch = getchar();
   }
   word[pos] = '\0';

   /* Return length of word. */
   return pos;
}
```

Continued on next slide

Appendix: The "justify" Program

```
void ClearLine(char *line, int *lineLen, int *numWords) {

/* Clear the given line.  That is, clear line, and set *lineLen
   and *numWords to 0. */

   line[0] = '\0';
   *lineLen = 0;
   *numWords = 0;
}

void AddWord(const char *word, char *line, int *lineLen) {

/* Append word to line, making sure that the words within line are
   separated with spaces.  Update *lineLen to indicate the
   new line length. */

   /* If line already contains some words, append a space. */
   if (*lineLen > 0) {
      line[*lineLen] = ' ';
      line[*lineLen + 1] = '\0';
      (*lineLen)++;
   }
   strcat(line, word);
   (*lineLen) += strlen(word);
}
```

Continued on next slide

Appendix: The "justify" Program

```
void WriteLine(const char *line, int lineLen, int numWords) {

/* Write line to stdout, in right justified form.  lineLen
   indicates the number of characters in line.  numWords indicates
   the number of words in line. */

   int extraSpaces, spacesToInsert, i, j;

   /* Compute number of excess spaces for line. */
   extraSpaces = MAX_LINE_LEN - lineLen;

   for (i = 0; i < lineLen; i++) {
      if (line[i] != ' ')
         putchar(line[i]);
      else {
         /* Compute additional spaces to insert. */
         spacesToInsert = extraSpaces / (numWords - 1);

         /* Print a space, plus additional spaces. */
         for (j = 1; j <= spacesToInsert + 1; j++)
            putchar(' ');

         /* Decrease extra spaces and word count. */
         extraSpaces -= spacesToInsert;
         numWords--;
      }
   }
   putchar('\n');
}
```

Continued on next slide

Appendix: The "justify" Program

```
int main(void) {

/* Read words from stdin, and write the words in justified format
   to stdout. */

/* Simplifying assumptions:
   Each word ends with a space, tab, newline, or end-of-file.
   No word is longer than MAX_WORD_LEN characters. */

   char word[MAX_WORD_LEN + 1];
   int wordLen;

   char line[MAX_LINE_LEN + 1];
   int lineLen = 0;
   int numWords = 0;

   ClearLine(line, &lineLen, &numWords);

   …
```

Continued on next slide

Appendix: The "justify" Program

```
   …

   for (;;) {
      wordLen = ReadWord(word);

      /* If no more words, print line
         with no justification. */
      if ((wordLen == 0) && (lineLen > 0)) {
         puts(line);
         break;
      }

      /* If word doesn't fit on this line, then... */
      if ((wordLen + 1 + lineLen) > MAX_LINE_LEN) {
         WriteLine(line, lineLen, numWords);
         ClearLine(line, &lineLen, &numWords);
      }

      AddWord(word, line, &lineLen);
      numWords++;
   }
   return 0;
}
```

Data Structures and Algorithms

Jennifer Rexford

The material for this lecture is drawn, in part, from
The Practice of Programming (Kernighan & Pike) Chapter 2

Motivating Quotations

"Every program depends on algorithms and data structures, but few programs depend on the invention of brand new ones."

-- Kernighan & Pike

"I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his *code* or his *data structures* more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships."

-- Linus Torvalds

Goals of this Lecture

- Help you learn (or refresh your memory) about:
  - Common data structures and algorithms
- Why? Shallow motivation:
  - Provide examples of pointer-related C code
- Why? Deeper motivation:
  - Common data structures and algorithms serve as "high level building blocks"
  - A power programmer:
    - Rarely creates programs from scratch
    - Often creates programs using building blocks

A Common Task

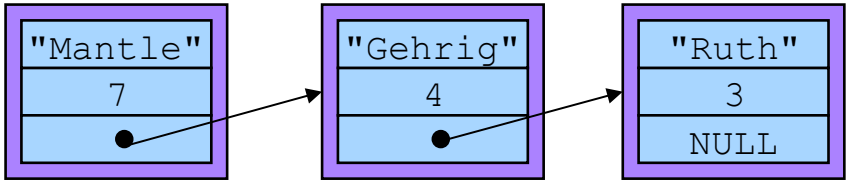
- Maintain a table of key/value pairs
  - Each key is a string; each value is an **`int`**
  - Unknown number of key-value pairs
- Examples
  - (student name, grade)
    - (“john smith”, 84), (“jane doe”, 93), (“bill clinton”, 81)
  - (baseball player, number)
    - (“Ruth”, 3), (“Gehrig”, 4), (“Mantle”, 7)
  - (variable name, value)
    - (“maxLength”, 2000), (“i”, 7), (“j”, -10)
- For simplicity, allow duplicate keys (client responsibility)
  - In Assignment #3, must check for duplicate keys!

Data Structures and Algorithms

- **Data structures**
  - Linked list of key/value pairs
  - Hash table of key/value pairs
- **Algorithms**
  - Create: Create the data structure
  - Add: Add a key/value pair
  - Search: Search for a key/value pair, by key
  - Free: Free the data structure

Data Structure #1: Linked List

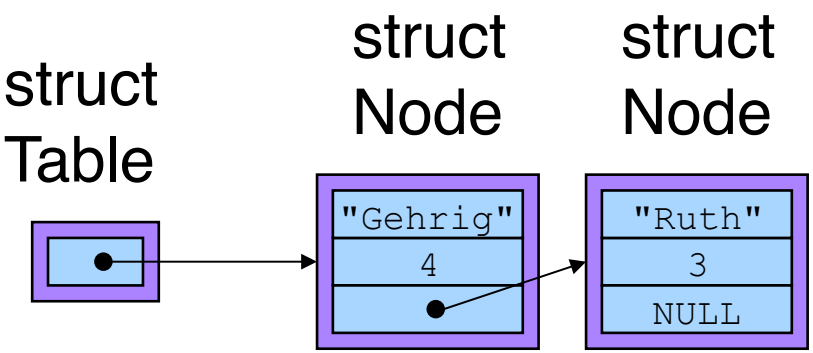
- Data structure: Nodes; each contains key/value pair and pointer to next node



- Algorithms:
  - Create: Allocate Table structure to point to first node
  - Add: Insert new node at front of list
  - Search: Linear search through the list
  - Free: Free nodes while traversing; free Table structure

Linked List: Data Structure

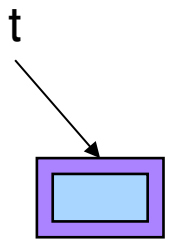
```
struct Node {
   const char *key;
   int value;
   struct Node *next;
};

struct Table {
   struct Node *first;
};
```



Linked List: Create (1)

```
struct Table *Table_create(void) {
   struct Table *t;
   t = (struct Table*)
      malloc(sizeof(struct Table));
   t->first = NULL;
   return t;
}
```

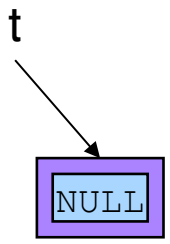
```
struct Table *t;
…
t = Table_create();
…
```



Linked List: Create (2)

```
struct Table *Table_create(void) {
    struct Table *t;
    t = (struct Table*)
        malloc(sizeof(struct Table));
    t->first = NULL;
    return t;
}
```

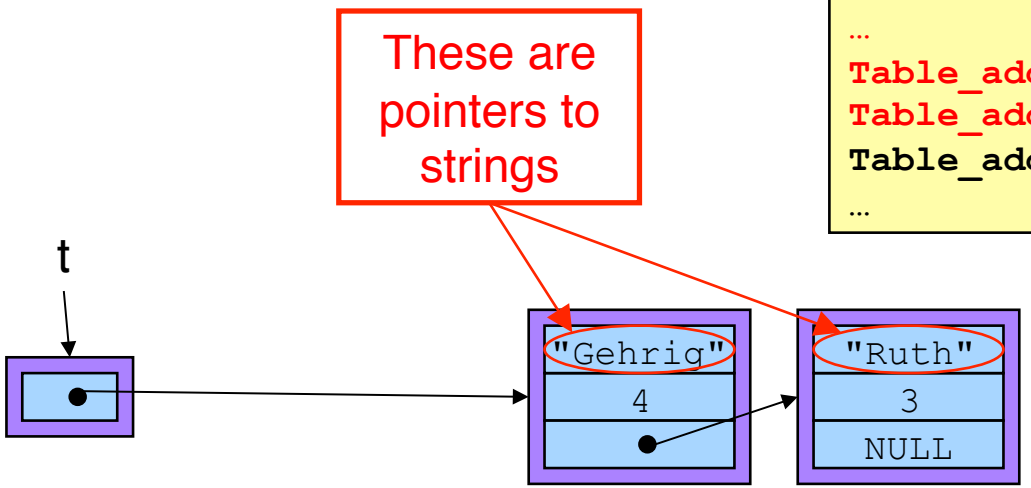
```
struct Table *t;
…
t = Table_create();
…
```



Linked List: Add (1)

```
void Table_add(struct Table *t,
    const char *key, int value) {
    struct Node *p = (struct Node*)malloc(sizeof(struct Node));
    p->key = key;
    p->value = value;
    p->next = t->first;
    t->first = p;
}
```

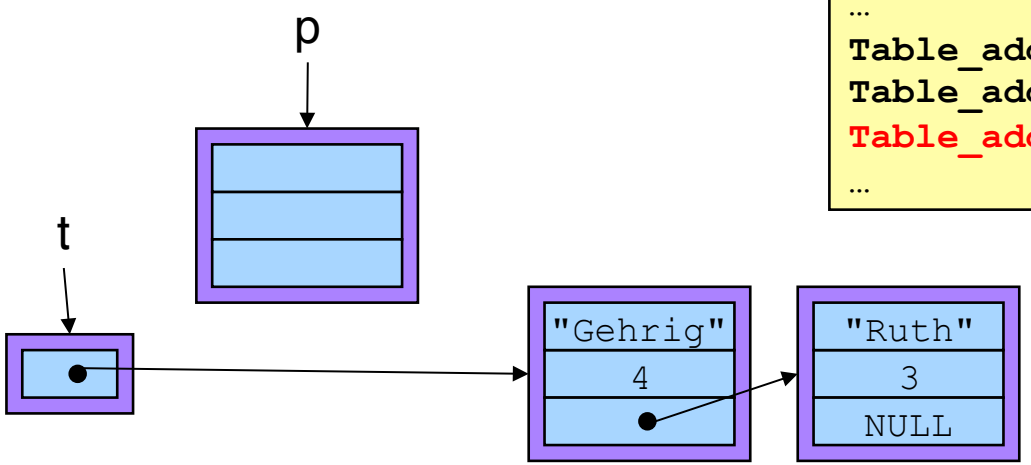
```
struct Table *t;
…
Table_add(t, "Ruth", 3);
Table_add(t, "Gehrig", 4);
Table_add(t, "Mantle", 7);
…
```



Linked List: Add (2)

```
void Table_add(struct Table *t,
    const char *key, int value) {
    struct Node *p = (struct Node*)malloc(sizeof(struct Node));
    p->key = key;
    p->value = value;
    p->next = t->first;
    t->first = p;
}
```

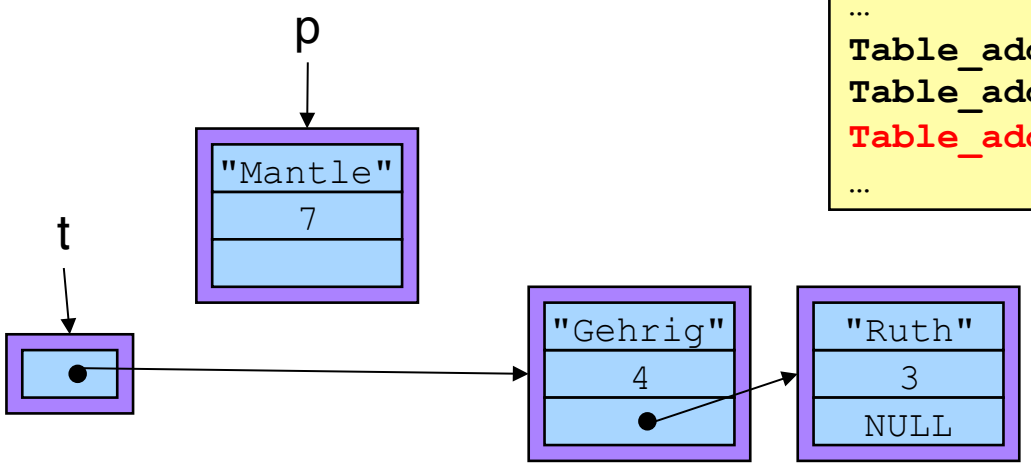
```
struct Table *t;
…
Table_add(t, "Ruth", 3);
Table_add(t, "Gehrig", 4);
Table_add(t, "Mantle", 7);
…
```



Linked List: Add (3)

```
void Table_add(struct Table *t,
    const char *key, int value) {
    struct Node *p = (struct Node*)malloc(sizeof(struct Node));
    p->key = key;
    p->value = value;
    p->next = t->first;
    t->first = p;
}
```

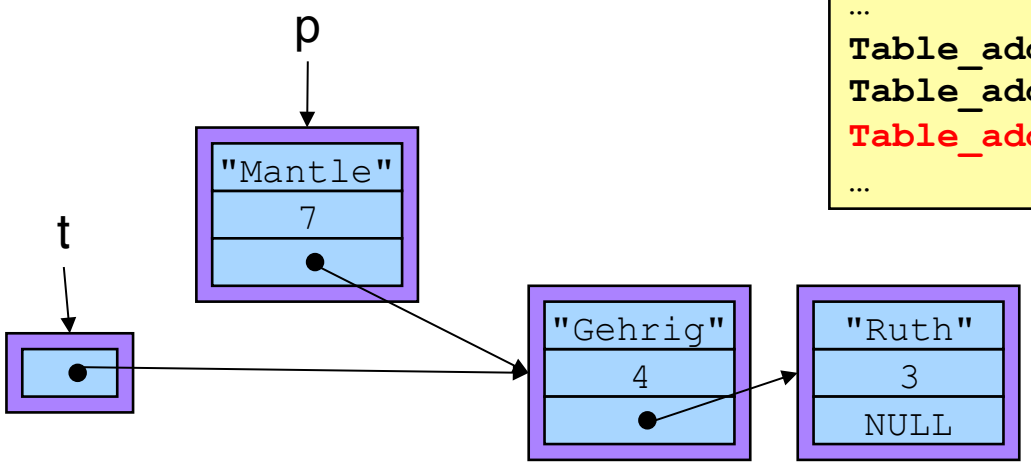
```
struct Table *t;
…
Table_add(t, "Ruth", 3);
Table_add(t, "Gehrig", 4);
Table_add(t, "Mantle", 7);
…
```



Linked List: Add (4)

```
void Table_add(struct Table *t,
    const char *key, int value) {
    struct Node *p = (struct Node*)malloc(sizeof(struct Node));
    p->key = key;
    p->value = value;
    p->next = t->first;
    t->first = p;
}
```

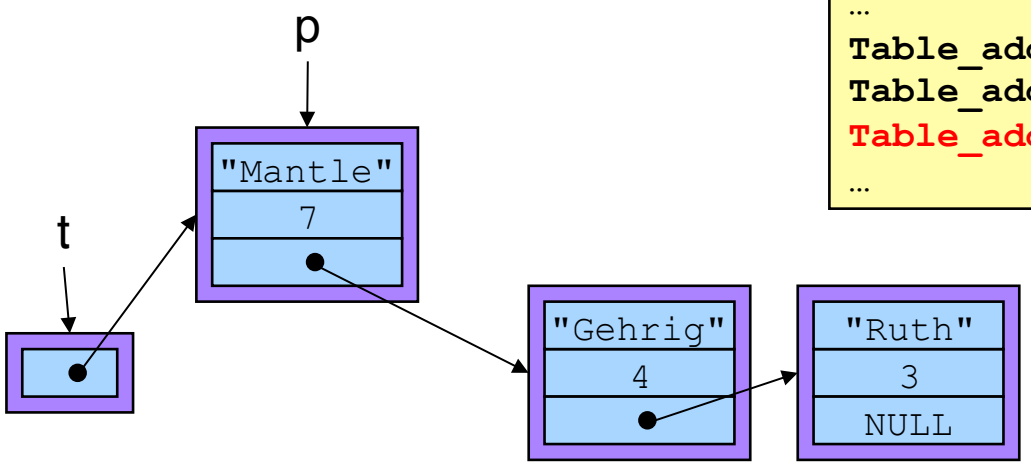
```
struct Table *t;
…
Table_add(t, "Ruth", 3);
Table_add(t, "Gehrig", 4);
Table_add(t, "Mantle", 7);
…
```



Linked List: Add (5)

```
void Table_add(struct Table *t,
    const char *key, int value) {
    struct Node *p = (struct Node*)malloc(sizeof(struct Node));
    p->key = key;
    p->value = value;
    p->next = t->first;
    t->first = p;
}
```

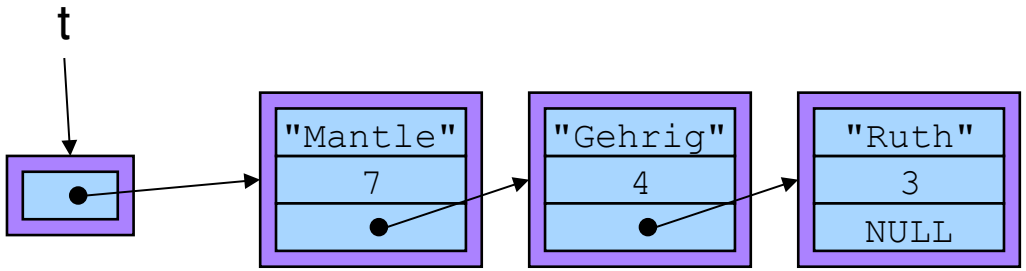
```
struct Table *t;
…
Table_add(t, "Ruth", 3);
Table_add(t, "Gehrig", 4);
Table_add(t, "Mantle", 7);
…
```



Linked List: Search (1)

```
int Table_search(struct Table *t,
    const char *key, int *value) {
    struct Node *p;
    for (p = t->first; p != NULL; p = p->next)
        if (strcmp(p->key, key) == 0) {
            *value = p->value;
            return 1;
        }
    return 0;
}
```

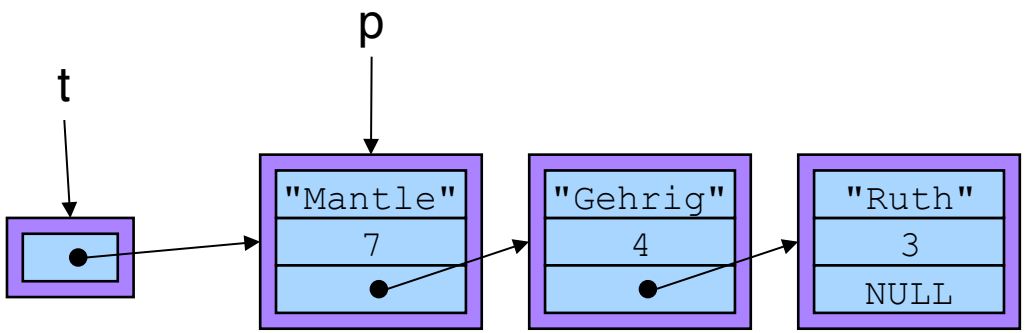
```
struct Table *t;
int value;
int found;
…
found =
    Table_search(t, "Gehrig", &value);
…
```



Linked List: Search (2)

```
int Table_search(struct Table *t,
    const char *key, int *value) {
    struct Node *p;
    for (p = t->first; p != NULL; p = p->next)
        if (strcmp(p->key, key) == 0) {
            *value = p->value;
            return 1;
        }
    return 0;
}
```

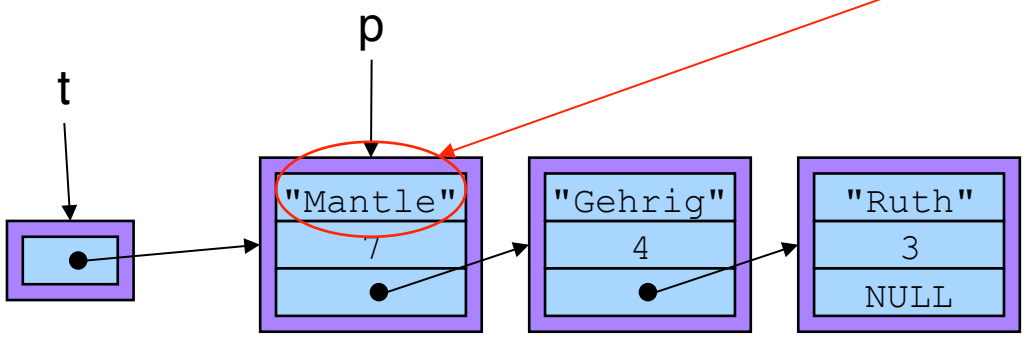
```
struct Table *t;
int value;
int found;
…
found =
    Table_search(t, "Gehrig", &value);
…
```



Linked List: Search (3)

```
int Table_search(struct Table *t,
    const char *key, int *value) {
    struct Node *p;
    for (p = t->first; p != NULL; p = p->next)
        if (strcmp(p->key, key) == 0) {
            *value = p->value;
            return 1;
        }
    return 0;
}
```

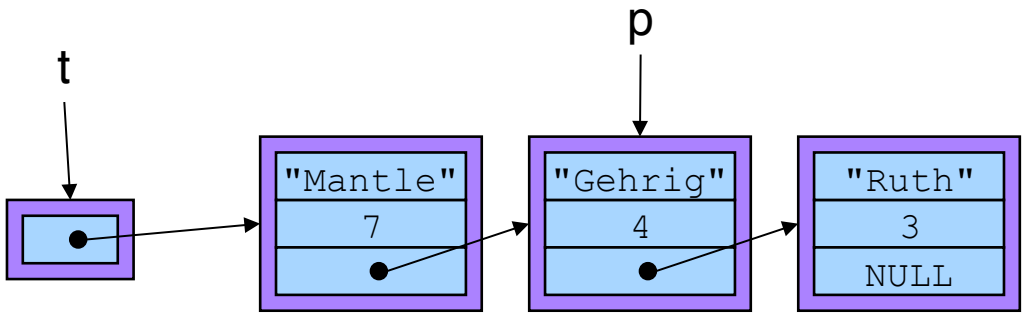
```
struct Table *t;
int value;
int found;
…
found =
    Table_search(t, "Gehrig", &value);
…
```



Linked List: Search (4)

```
int Table_search(struct Table *t,
    const char *key, int *value) {
    struct Node *p;
    for (p = t->first; p != NULL; p = p->next)
        if (strcmp(p->key, key) == 0) {
            *value = p->value;
            return 1;
        }
    return 0;
}
```

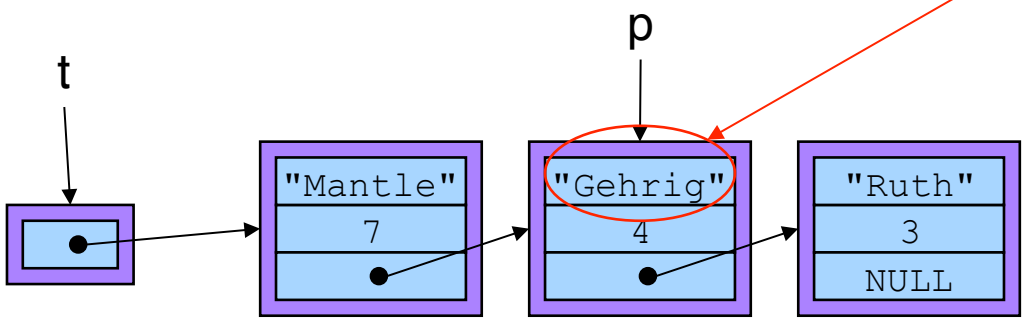
```
struct Table *t;
int value;
int found;
…
found =
    Table_search(t, "Gehrig", &value);
…
```



Linked List: Search (5)

```
int Table_search(struct Table *t,
    const char *key, int *value) {
    struct Node *p;
    for (p = t->first; p != NULL; p = p->next)
        if (strcmp(p->key, key) == 0) {
            *value = p->value;
            return 1;
        }
    return 0;
}
```

```
struct Table *t;
int value;
int found;
…
found =
    Table_search(t, "Gehrig", &value);
…
```



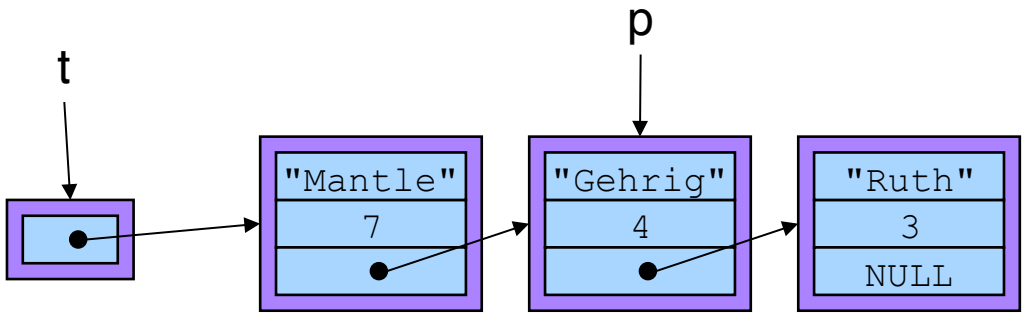
Linked List: Search (6)

```
int Table_search(struct Table *t,
    const char *key, int *value) {
    struct Node *p;
    for (p = t->first; p != NULL; p = p->next)
        if (strcmp(p->key, key) == 0) {
            *value = p->value;
            return 1;
        }
    return 0;
}
```

```
struct Table *t;
int value;
int found;
…
found =
    Table_search(t, "Gehrig", &value);
…
```

1

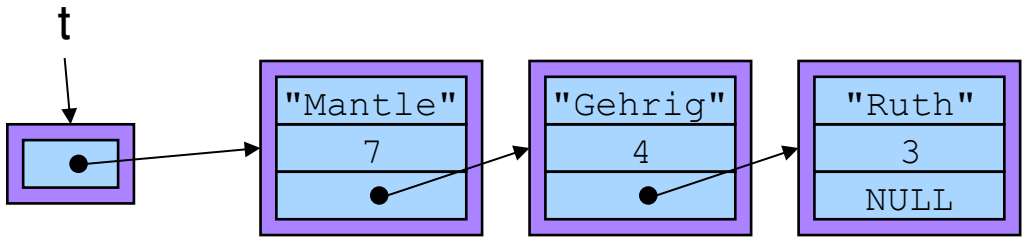
4



Linked List: Free (1)

```
void Table_free(struct Table *t) {
    struct Node *p;
    struct Node *nextp;
    for (p = t->first; p != NULL; p = nextp) {
        nextp = p->next;
        free(p);
    }
    free(t);
}
```

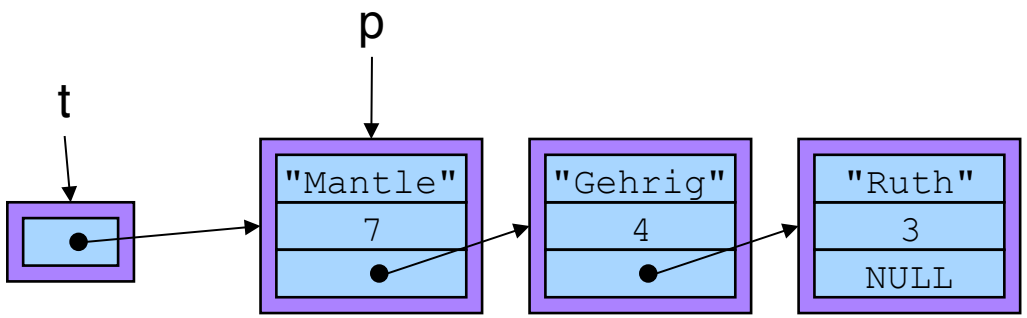
```
struct Table *t;
…
Table_free(t);
…
```



Linked List: Free (2)

```
void Table_free(struct Table *t) {
    struct Node *p;
    struct Node *nextp;
    for (p = t->first; p != NULL; p = nextp) {
        nextp = p->next;
        free(p);
    }
    free(t);
}
```

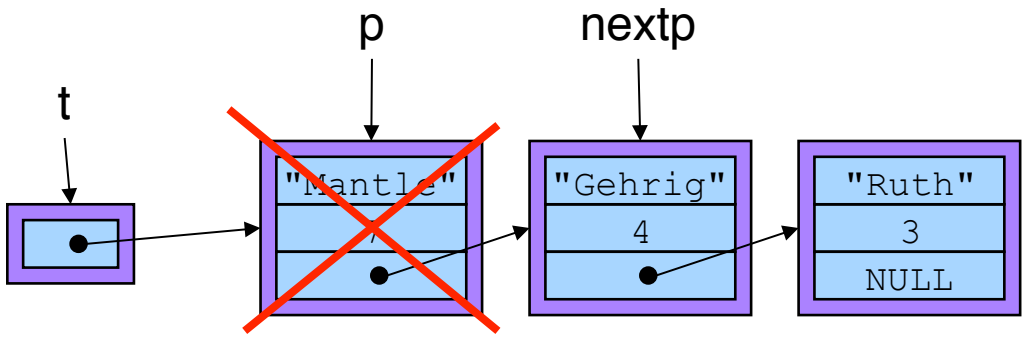
```
struct Table *t;
…
Table_free(t);
…
```



Linked List: Free (3)

```
void Table_free(struct Table *t) {
    struct Node *p;
    struct Node *nextp;
    for (p = t->first; p != NULL; p = nextp) {
        nextp = p->next;
        free(p);
    }
    free(t);
}
```

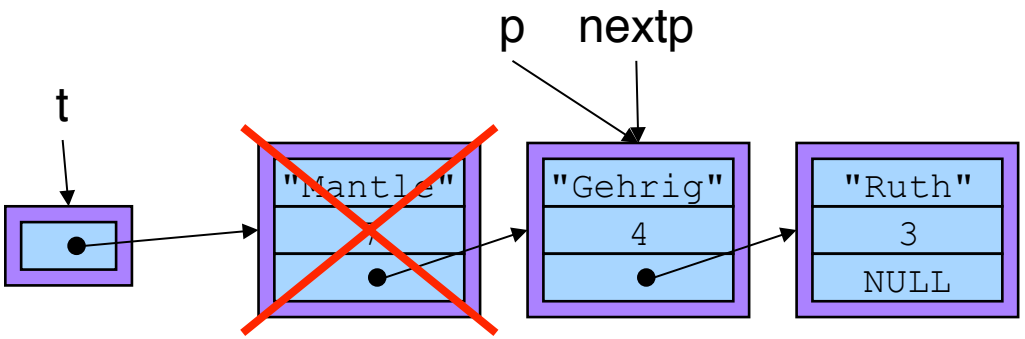
```
struct Table *t;
…
Table_free(t);
…
```



Linked List: Free (4)

```
void Table_free(struct Table *t) {
    struct Node *p;
    struct Node *nextp;
    for (p = t->first; p != NULL; p = nextp) {
        nextp = p->next;
        free(p);
    }
    free(t);
}
```

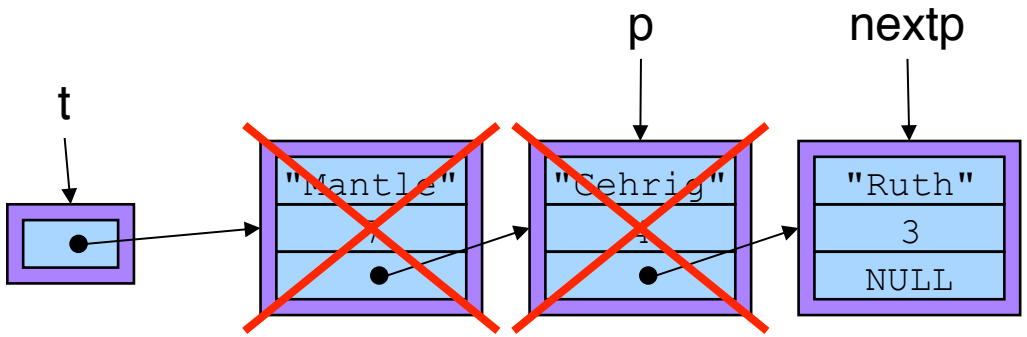
```
struct Table *t;
…
Table_free(t);
…
```



Linked List: Free (5)

```
void Table_free(struct Table *t) {
    struct Node *p;
    struct Node *nextp;
    for (p = t->first; p != NULL; p = nextp) {
        nextp = p->next;
        free(p);
    }
    free(t);
}
```

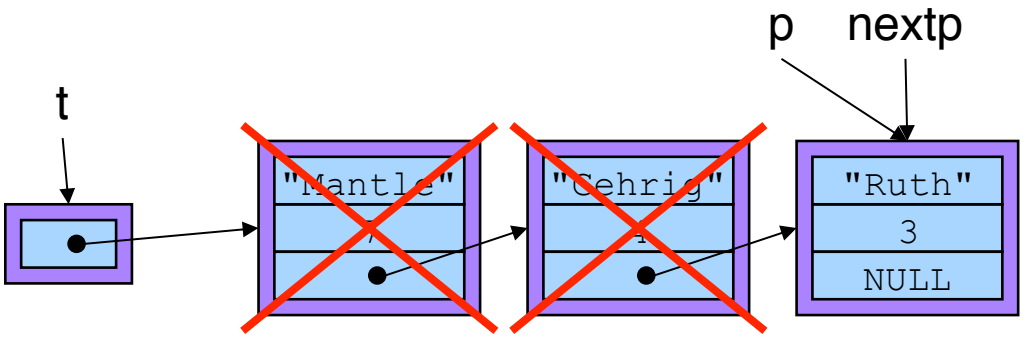
```
struct Table *t;
…
Table_free(t);
…
```



Linked List: Free (6)

```
void Table_free(struct Table *t) {
    struct Node *p;
    struct Node *nextp;
    for (p = t->first; p != NULL; p = nextp) {
        nextp = p->next;
        free(p);
    }
    free(t);
}
```

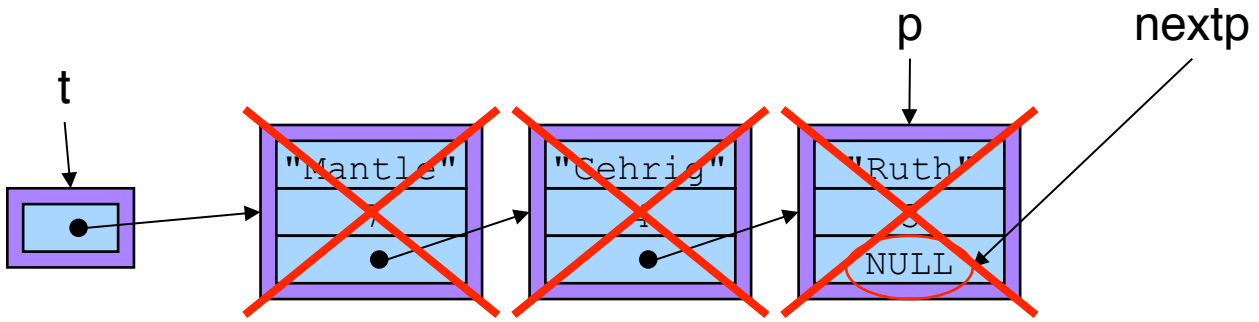
```
struct Table *t;
…
Table_free(t);
…
```



Linked List: Free (7)

```
void Table_free(struct Table *t) {
    struct Node *p;
    struct Node *nextp;
    for (p = t->first; p != NULL; p = nextp) {
        nextp = p->next;
        free(p);
    }
    free(t);
}
```

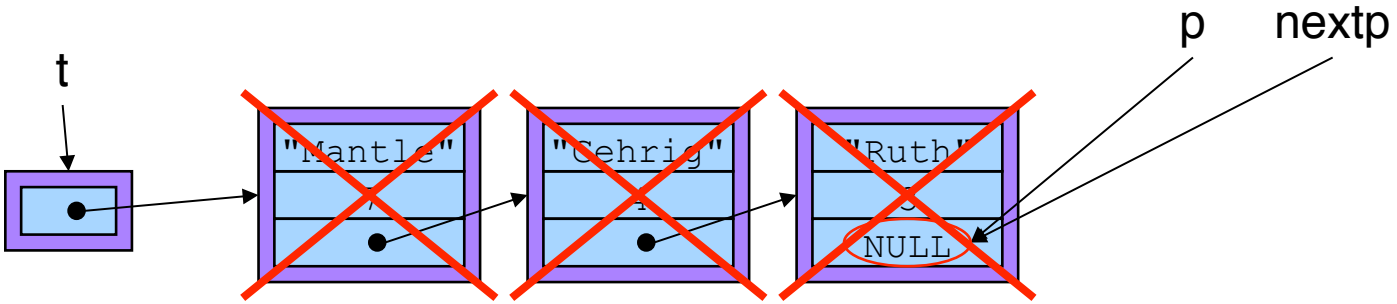
```
struct Table *t;
…
Table_free(t);
…
```



Linked List: Free (8)

```
void Table_free(struct Table *t) {
    struct Node *p;
    struct Node *nextp;
    for (p = t->first; p != NULL; p = nextp) {
        nextp = p->next;
        free(p);
    }
    free(t);
}
```

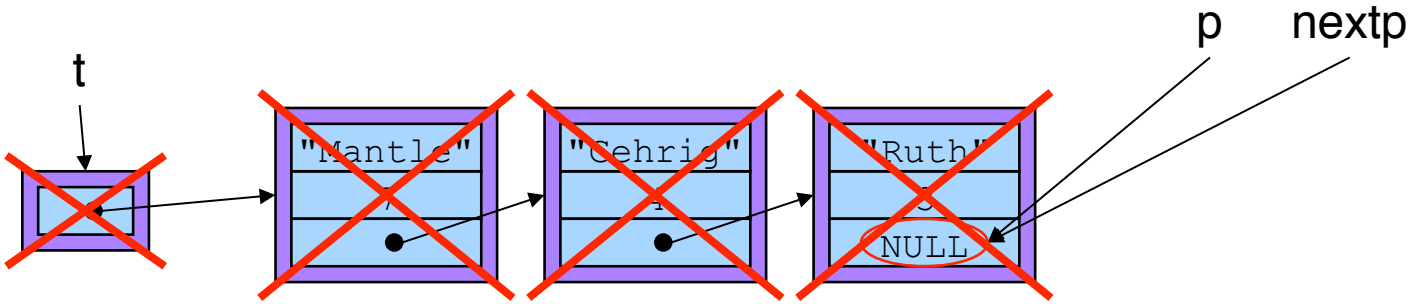
```
struct Table *t;
…
Table_free(t);
…
```



Linked List: Free (9)

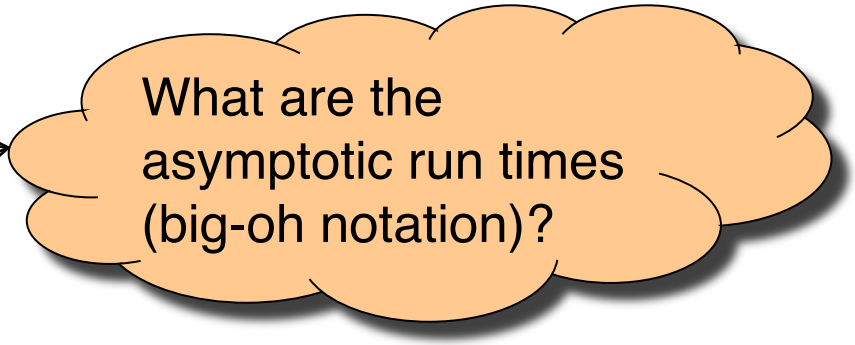
```
void Table_free(struct Table *t) {
    struct Node *p;
    struct Node *nextp;
    for (p = t->first; p != NULL; p = nextp) {
        nextp = p->next;
        free(p);
    }
    free(t)
}
```

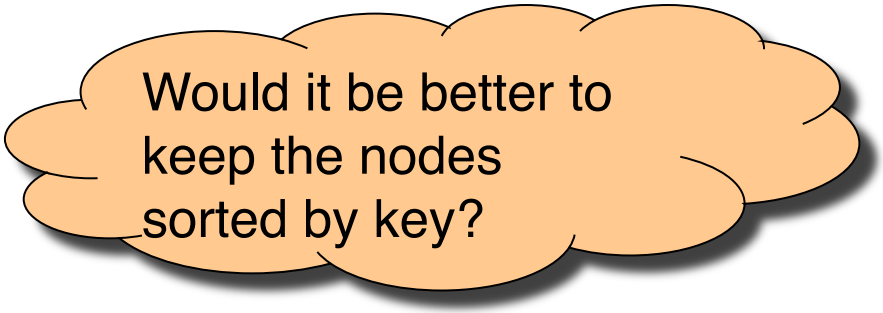
```
struct Table *t;
…
Table_free(t);
…
```



Linked List Performance

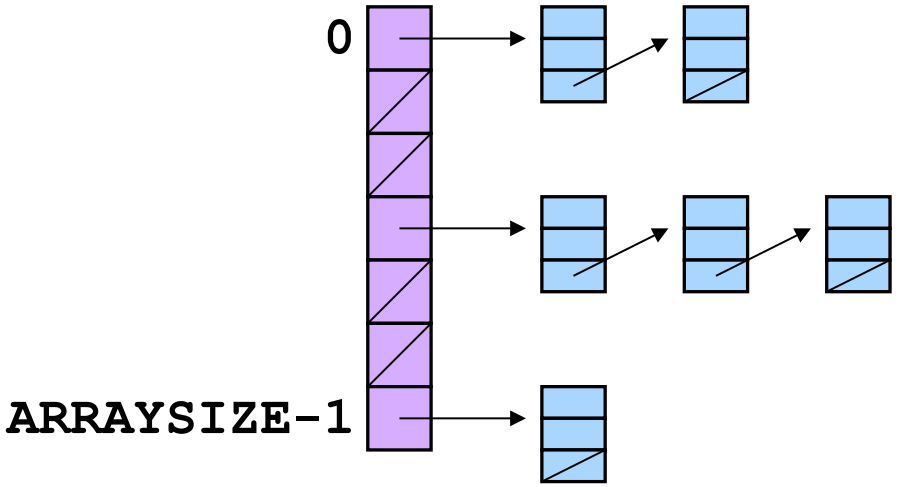
- Create: fast
- Add: fast
- Search: slow
- Free: slow





Data Structure #2: Hash Table

- Fixed-size array where each element points to a linked list

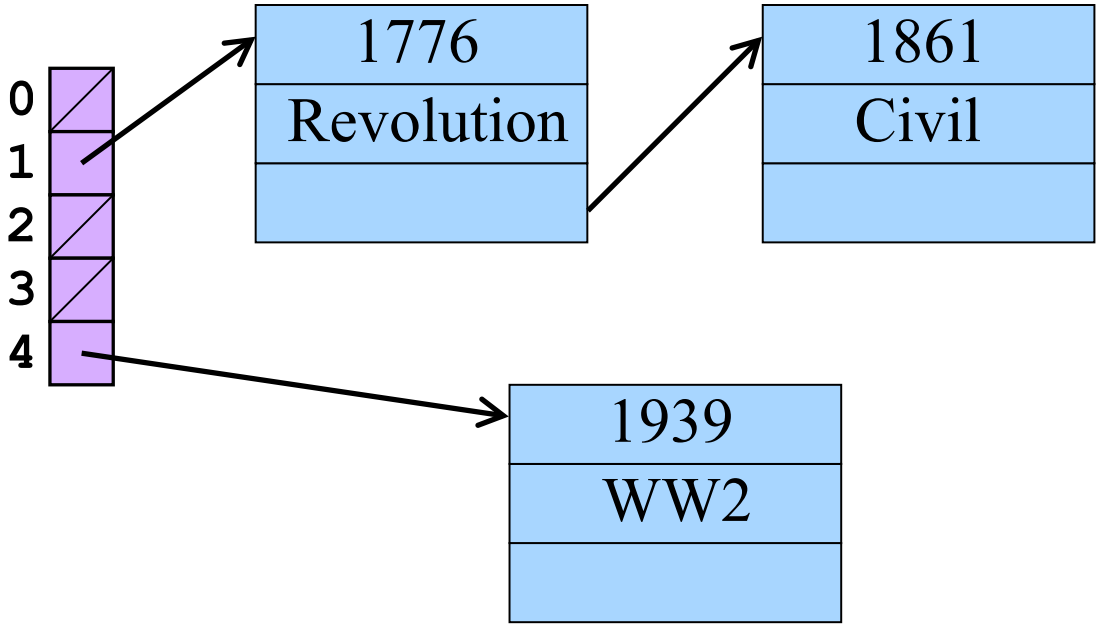



```
struct Node *array[ARRAYSIZE];
```

- Function maps each key to an array index
  - For example, for an integer key **`h`**
    - Hash function: **`i = h % ARRAYSIZE`** (mod function)
  - Go to array element **`i`**, i.e., the linked list **`hashtab[i]`**
    - Search for element, add element, remove element, etc.

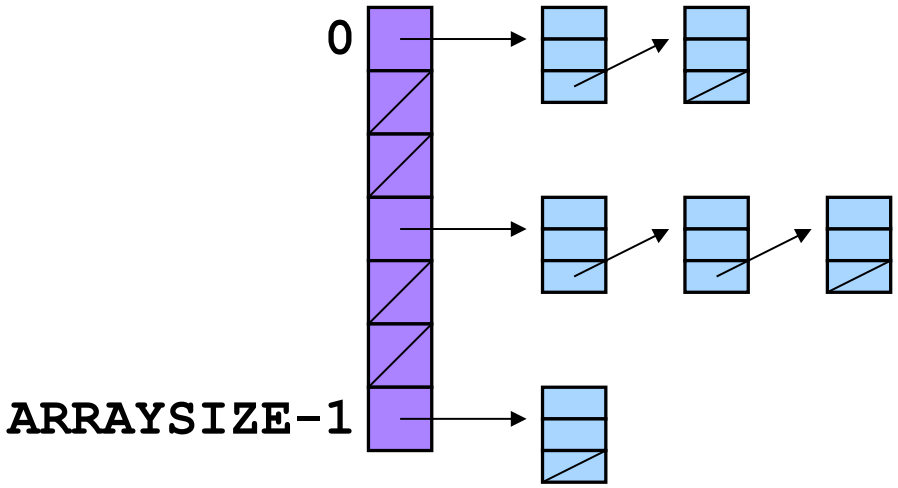
Hash Table Example

- Integer keys, array of size 5 with hash function “h mod 5”
  - “1776 % 5” is 1
  - “1861 % 5” is 1
  - “1939 % 5” is 4



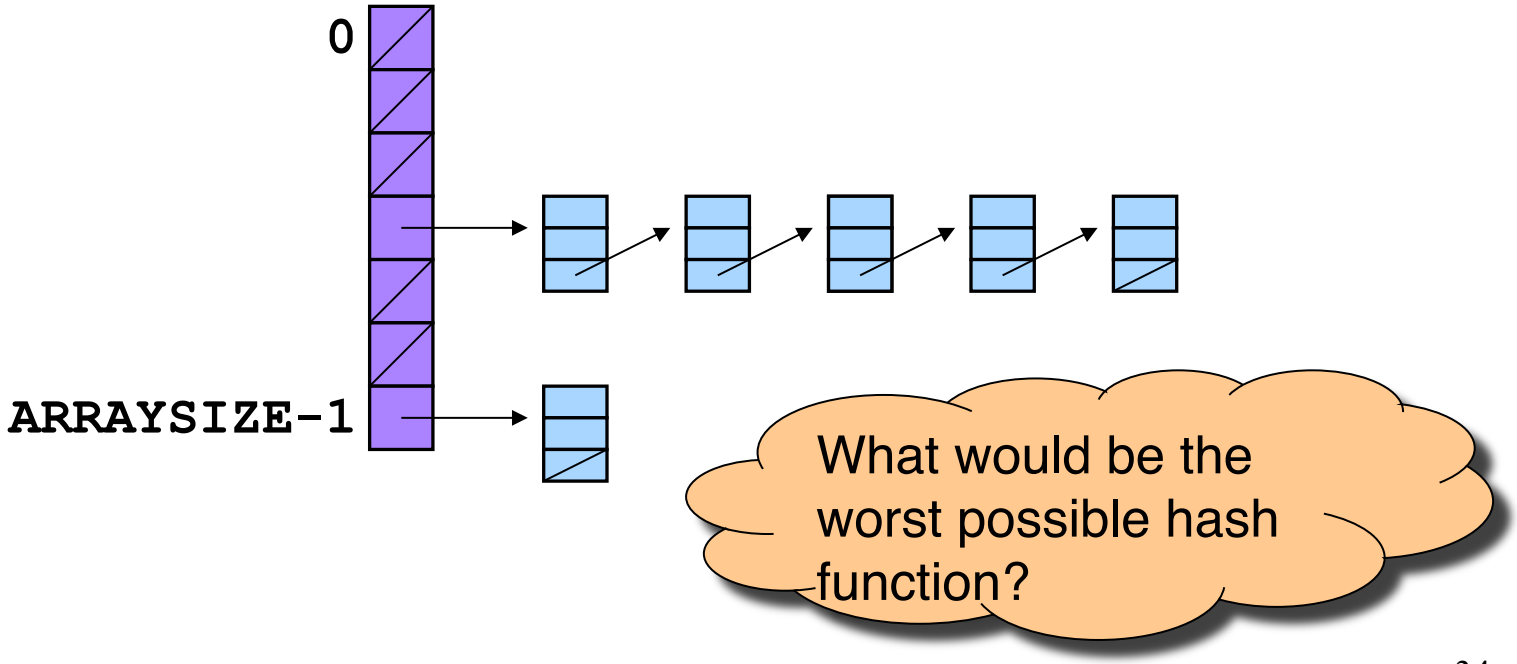
How Large an Array?

- Large enough that average "bucket" size is 1
  - Short buckets mean fast search
  - Long buckets mean slow search
- Small enough to be memory efficient
  - Not an excessive number of elements
  - Fortunately, each array element is just storing a pointer
- This is OK:



What Kind of Hash Function?

- Good at distributing elements across the array
  - Distribute results over the range 0, 1, …, ARRAYSIZE-1
  - Distribute results *evenly* to avoid very long buckets
- This is not so good:



Hashing String Keys to Integers

- Simple schemes don't distribute the keys evenly enough
  - Number of characters, mod ARRAYSIZE
  - Sum the ASCII values of all characters, mod ARRAYSIZE
  - …
- Here's a reasonably good hash function
  - Weighted sum of characters $\mathbf{x_i}$ in the string
    - $(\Sigma\ a^i x_i)$ mod ARRAYSIZE
  - Best if a and ARRAYSIZE are relatively prime
    - E.g., a = 65599, ARRAYSIZE = 1024

Implementing Hash Function

- Potentially expensive to compute a^i for each value of i
  - Computing a^i for each value of I
  - Instead, do (((x[0] * 65599 + x[1]) * 65599 + x[2]) * 65599 + x[3]) * …

```
unsigned int hash(const char *x) {
   int i;
   unsigned int h = 0U;
   for (i=0; x[i]!='\0'; i++)
      h = h * 65599 + (unsigned char)x[i];
   return h % 1024;
}
```

Can be more clever than this for powers of two!
(Described in Appendix)

Hash Table Example

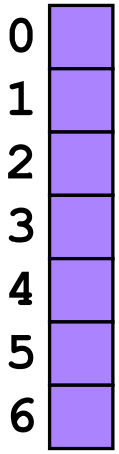
Example: ARRAYSIZE = 7

Lookup (and enter, if not present) these strings: the, cat, in, the, hat

Hash table initially empty.

First word: the. hash("the") = 965156977. 965156977 % 7 = 1.

Search the linked list table[1] for the string "the"; not found.



Hash Table Example (cont.)

Example: ARRAYSIZE = 7

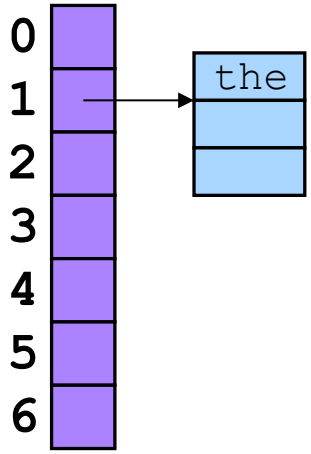
Lookup (and enter, if not present) these strings: the, cat, in, the, hat

Hash table initially empty.

First word: “the”. hash(“the”) = 965156977. 965156977 % 7 = 1.

Search the linked list table[1] for the string “the”; not found

Now: table[1] = makelink(key, value, table[1])

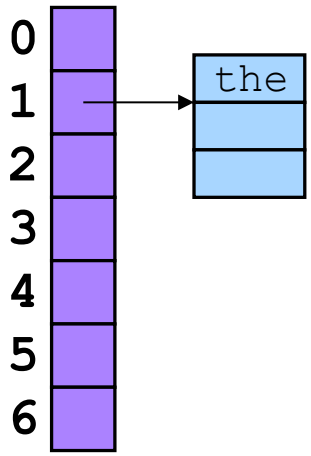


Hash Table Example (cont.)

Second word: “cat”. hash(“cat”) = 3895848756. 3895848756 % 7 = 2.

Search the linked list table[2] for the string “cat”; not found

Now: table[2] = makelink(key, value, table[2])

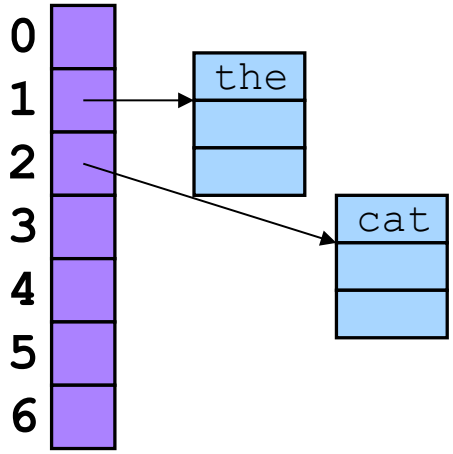


Hash Table Example (cont.)

Third word: “in”. hash(“in”) = 6888005. 6888005% 7 = 5.

Search the linked list table[5] for the string “in”; not found

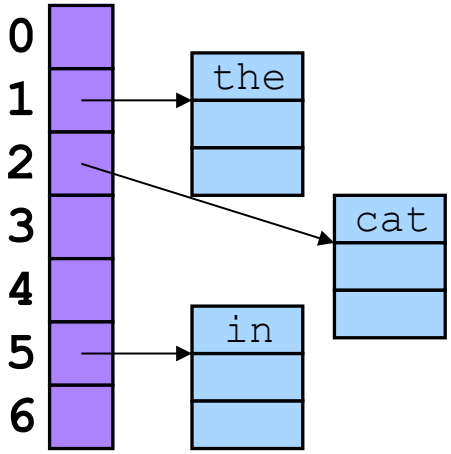
Now: table[5] = makelink(key, value, table[5])



Hash Table Example (cont.)

Fourth word: “the”. hash(“the”) = 965156977. 965156977 % 7 = 1.

Search the linked list table[1] for the string “the”; found it!



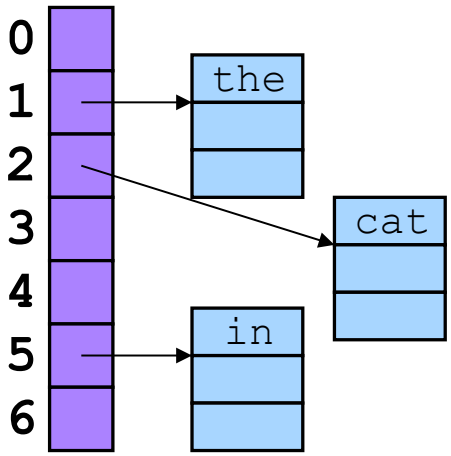
Hash Table Example (cont.)

Fourth word: “hat”. hash(“hat”) = 865559739. 865559739 % 7 = 2.

Search the linked list table[2] for the string “hat”; not found.

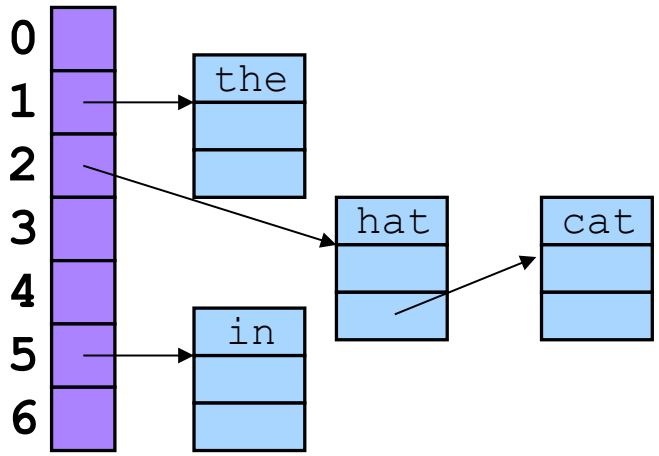
Now, insert “hat” into the linked list table[2].

At beginning or end? Doesn’t matter.



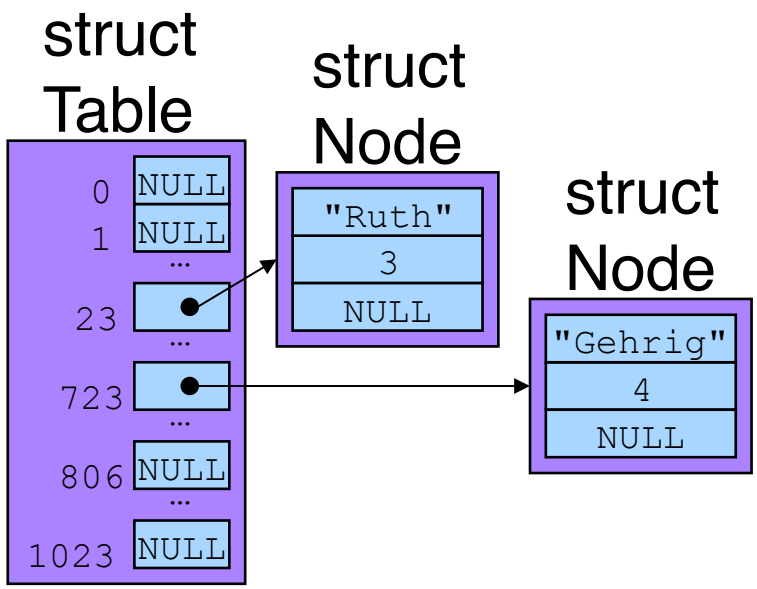
Hash Table Example (cont.)

Inserting at the front is easier, so add “hat” at the front



Hash Table: Data Structure

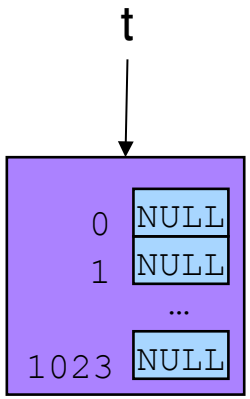
```
enum {BUCKET_COUNT = 1024};

struct Node {
   const char *key;
   int value;
   struct Node *next;
};

struct Table {
   struct Node *array[BUCKET_COUNT];
};
```



Hash Table: Create

```
struct Table *Table_create(void) {
   struct Table *t;
   t = (struct Table*)calloc(1, sizeof(struct Table));
   return t;
}
```

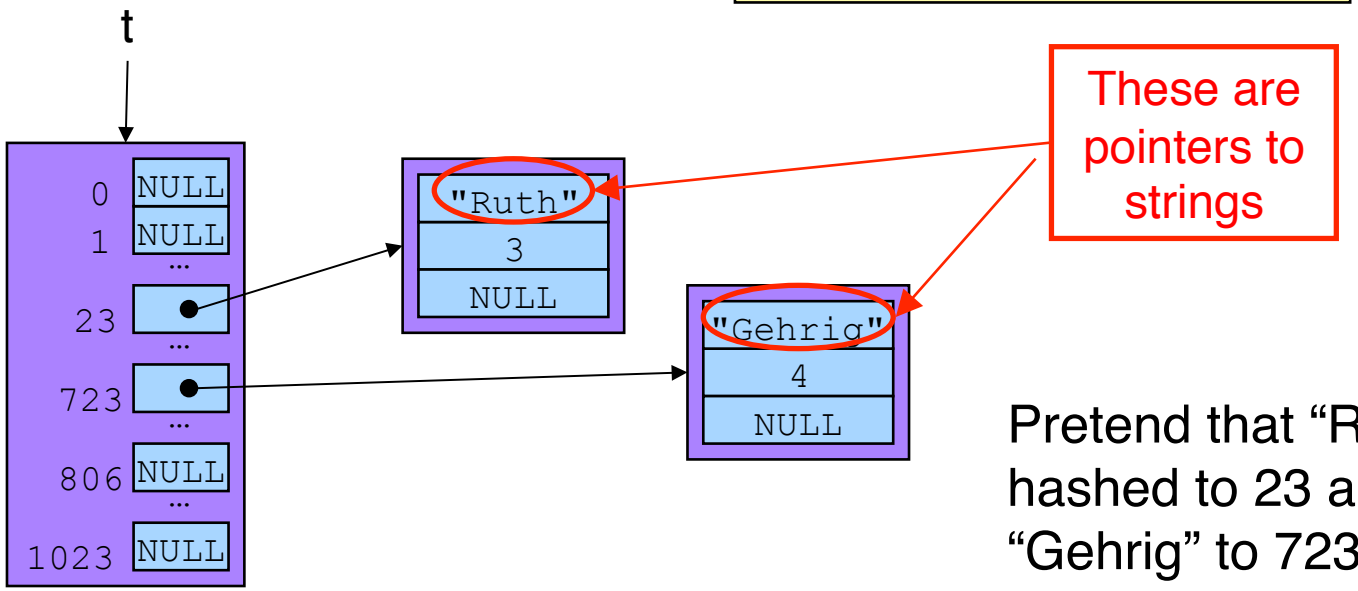
```
struct Table *t;
…
t = Table_create();
…
```



Hash Table: Add (1)

```
void Table_add(struct Table *t,
    const char *key, int value) {
    struct Node *p = (struct Node*)malloc(sizeof(struct Node));
    int h = hash(key);
    p->key = key;
    p->value = value;
    p->next = t->array[h];
    t->array[h] = p;
}
```

```
struct Table *t;
…
Table_add(t, "Ruth", 3);
Table_add(t, "Gehrig", 4);
Table_add(t, "Mantle", 7);
…
```

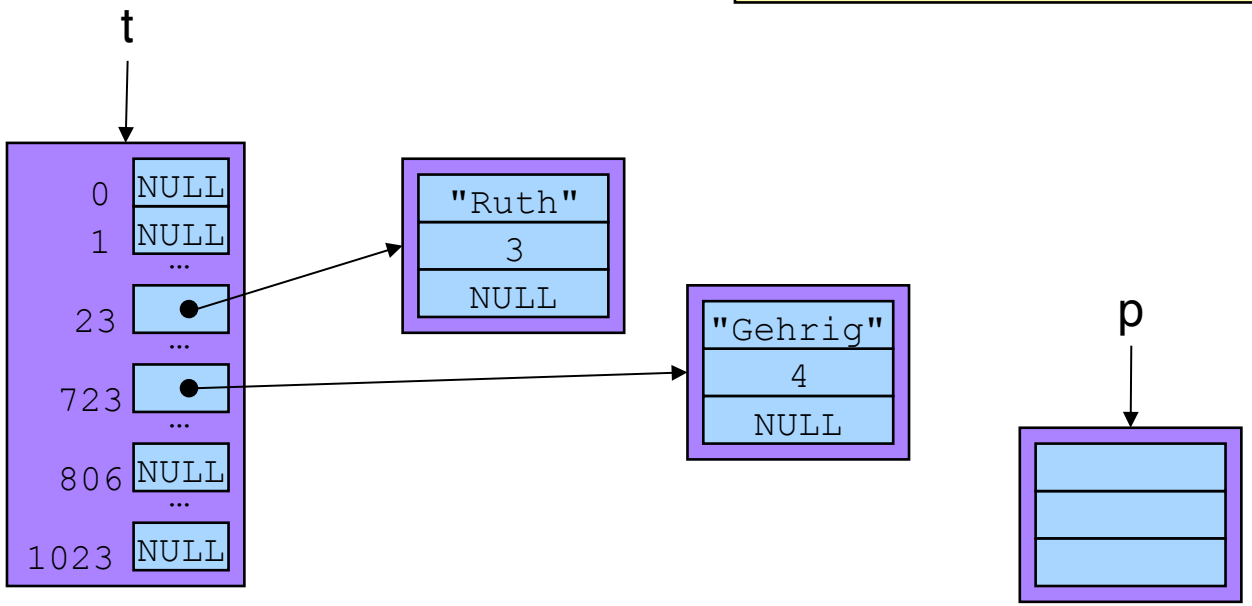


Pretend that "Ruth" hashed to 23 and "Gehrig" to 723

Hash Table: Add (2)

```
void Table_add(struct Table *t,
    const char *key, int value) {
    struct Node *p = (struct Node*)malloc(sizeof(struct Node));
    int h = hash(key);
    p->key = key;
    p->value = value;
    p->next = t->array[h];
    t->array[h] = p;
}
```

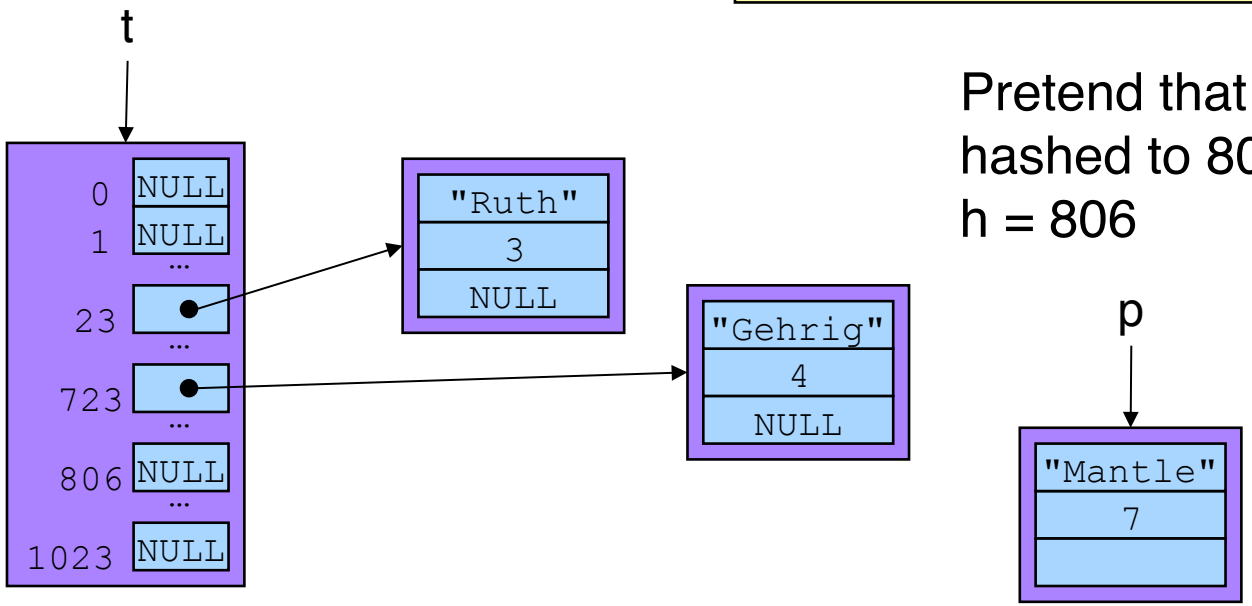
```
struct Table *t;
…
Table_add(t, "Ruth", 3);
Table_add(t, "Gehrig", 4);
Table_add(t, "Mantle", 7);
…
```



Hash Table: Add (3)

```
void Table_add(struct Table *t,
    const char *key, int value) {
    struct Node *p = (struct Node*)malloc(sizeof(struct Node));
    int h = hash(key);
    p->key = key;
    p->value = value;
    p->next = t->array[h];
    t->array[h] = p;
}
```

```
struct Table *t;
…
Table_add(t, "Ruth", 3);
Table_add(t, "Gehrig", 4);
Table_add(t, "Mantle", 7);
…
```

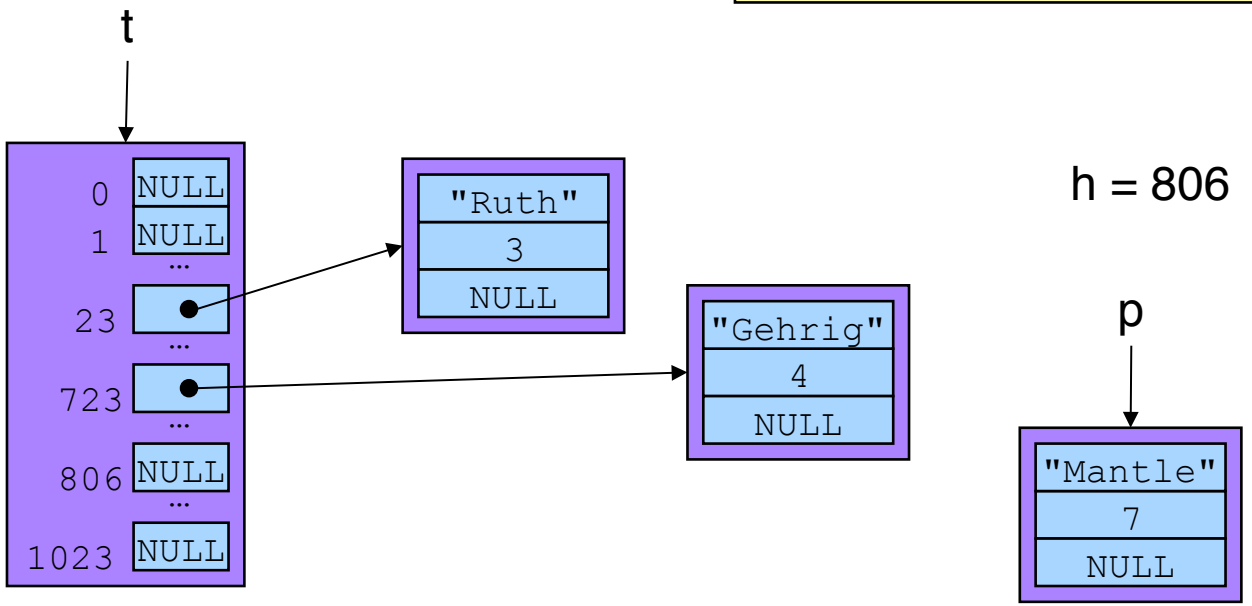


Pretend that “Mantle” hashed to 806, and so h = 806

Hash Table: Add (4)

```
void Table_add(struct Table *t,
    const char *key, int value) {
    struct Node *p = (struct Node*)malloc(sizeof(struct Node));
    int h = hash(key);
    p->key = key;
    p->value = value;
    p->next = t->array[h];
    t->array[h] = p;
}
```

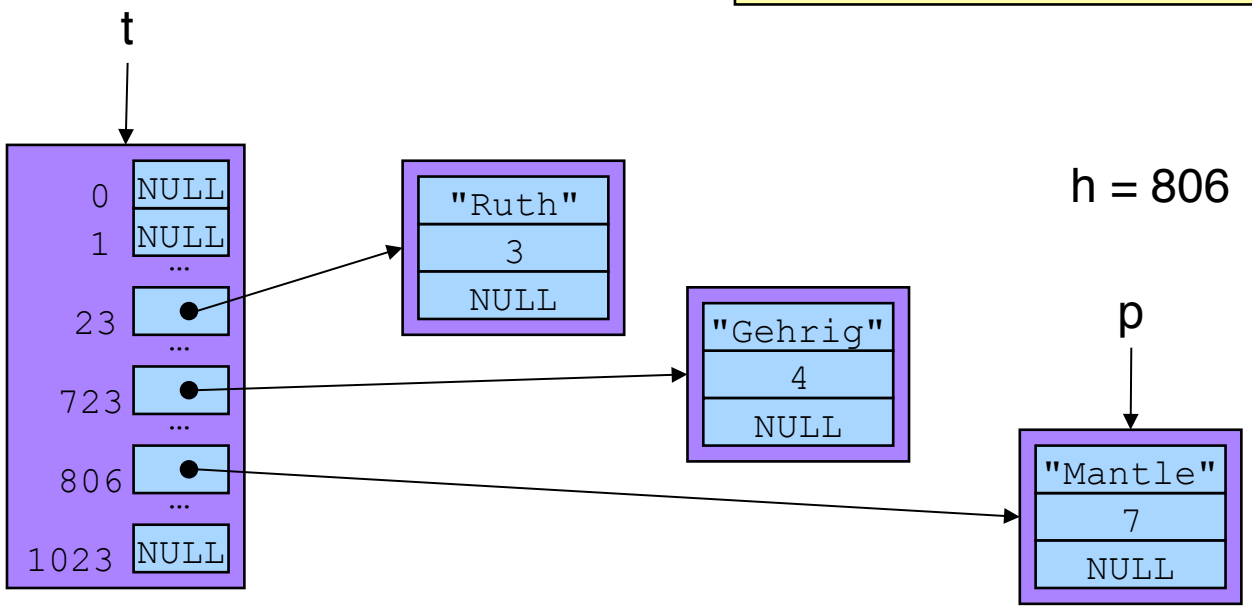
```
struct Table *t;
…
Table_add(t, "Ruth", 3);
Table_add(t, "Gehrig", 4);
Table_add(t, "Mantle", 7);
…
```



Hash Table: Add (5)

```
void Table_add(struct Table *t,
    const char *key, int value) {
    struct Node *p = (struct Node*)malloc(sizeof(struct Node));
    int h = hash(key);
    p->key = key;
    p->value = value;
    p->next = t->array[h];
    t->array[h] = p;
}
```

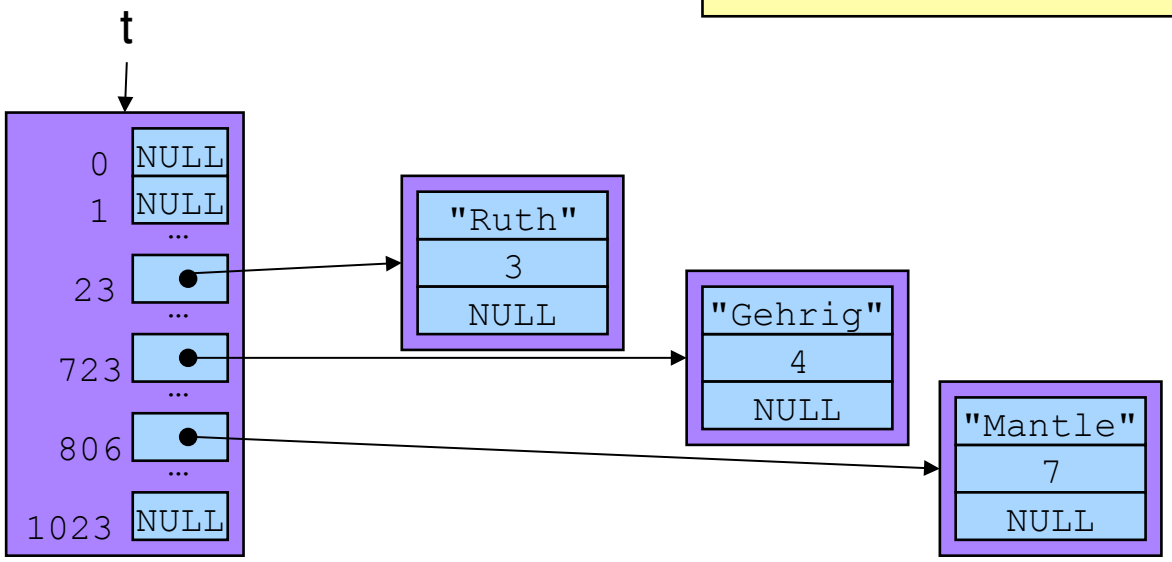
```
struct Table *t;
…
Table_add(t, "Ruth", 3);
Table_add(t, "Gehrig", 4);
Table_add(t, "Mantle", 7);
…
```



Hash Table: Search (1)

```
int Table_search(struct Table *t,
    const char *key, int *value) {
    struct Node *p;
    int h = hash(key);
    for (p = t->array[h]; p != NULL; p = p->next)
        if (strcmp(p->key, key) == 0) {
            *value = p->value;
            return 1;
        }
    return 0;
}
```

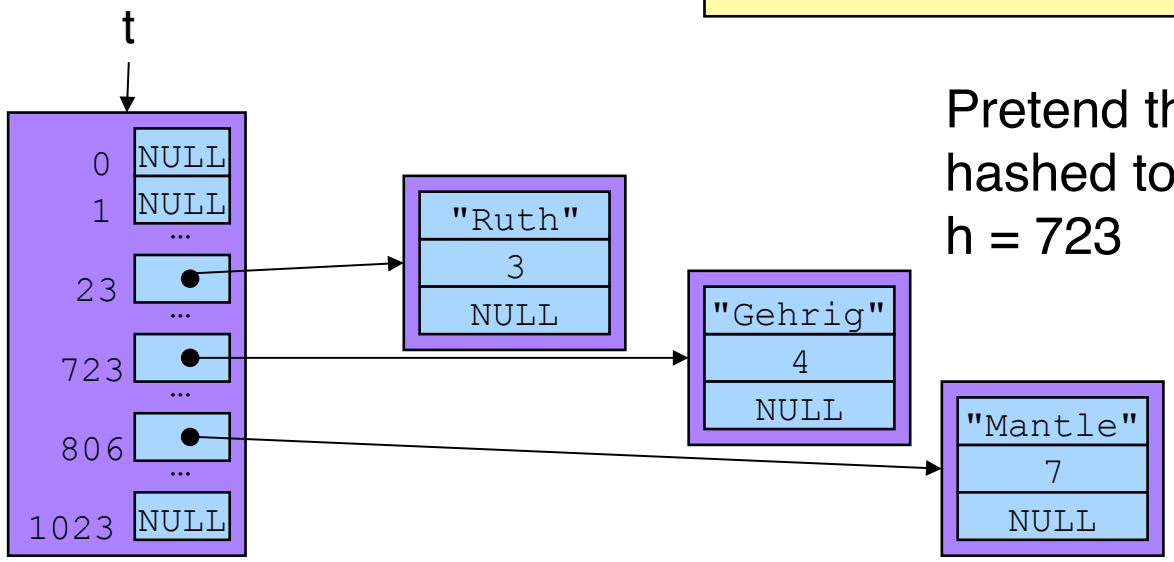
```
struct Table *t;
int value;
int found;
…
found =
    Table_search(t, "Gehrig", &value);
…
```



Hash Table: Search (2)

```
int Table_search(struct Table *t,
    const char *key, int *value) {
    struct Node *p;
    int h = hash(key);
    for (p = t->array[h]; p != NULL; p = p->next)
        if (strcmp(p->key, key) == 0) {
            *value = p->value;
            return 1;
        }
    return 0;
}
```

```
struct Table *t;
int value;
int found;
…
found =
    Table_search(t, "Gehrig", &value);
…
```

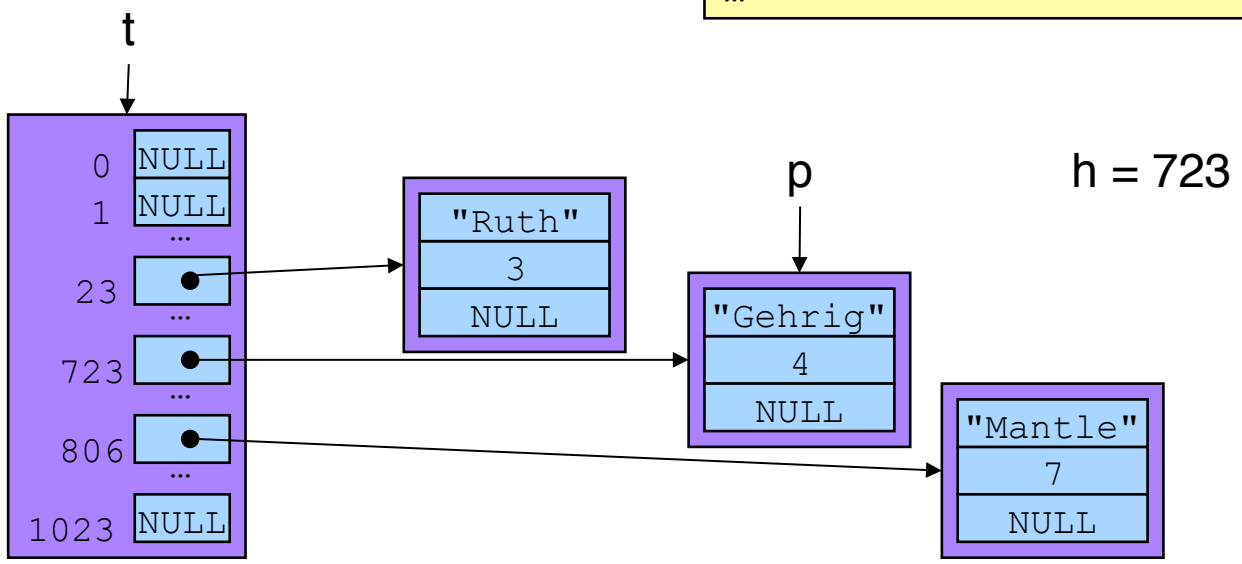


Pretend that “Gehrig” hashed to 723, and so h = 723

Hash Table: Search (3)

```
int Table_search(struct Table *t,
    const char *key, int *value) {
    struct Node *p;
    int h = hash(key);
    for (p = t->array[h]; p != NULL; p = p->next)
        if (strcmp(p->key, key) == 0) {
            *value = p->value;
            return 1;
        }
    return 0;
}
```

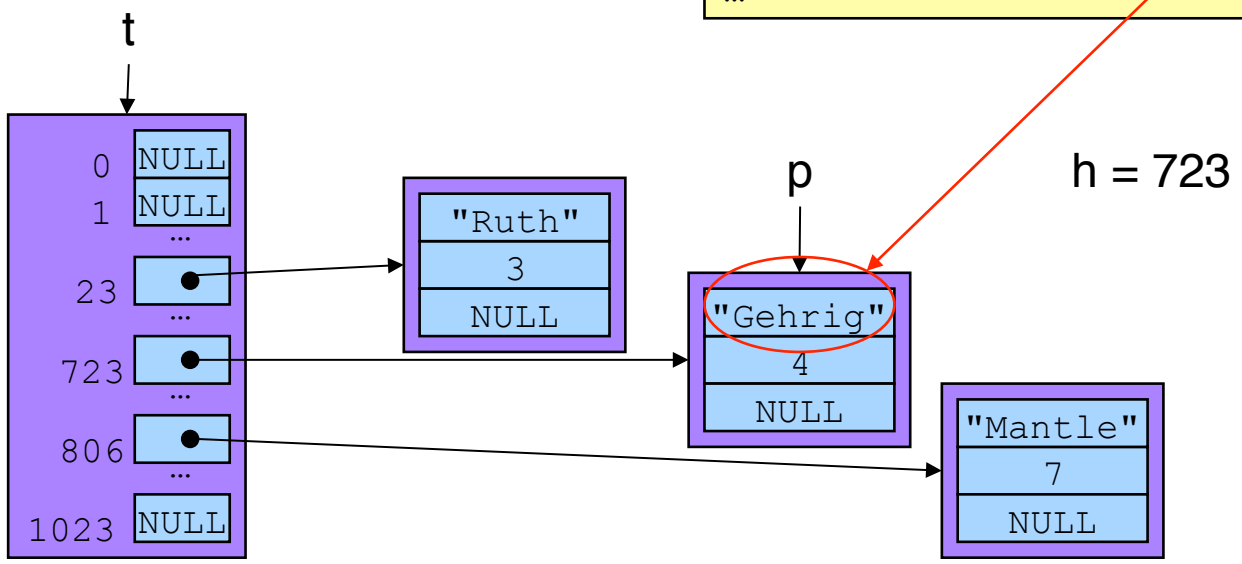
```
struct Table *t;
int value;
int found;
…
found =
    Table_search(t, "Gehrig", &value);
…
```



Hash Table: Search (4)

```
int Table_search(struct Table *t,
    const char *key, int *value) {
    struct Node *p;
    int h = hash(key);
    for (p = t->array[h]; p != NULL; p = p->next)
        if (strcmp(p->key, key) == 0) {
            *value = p->value;
            return 1;
        }
    return 0;
}
```

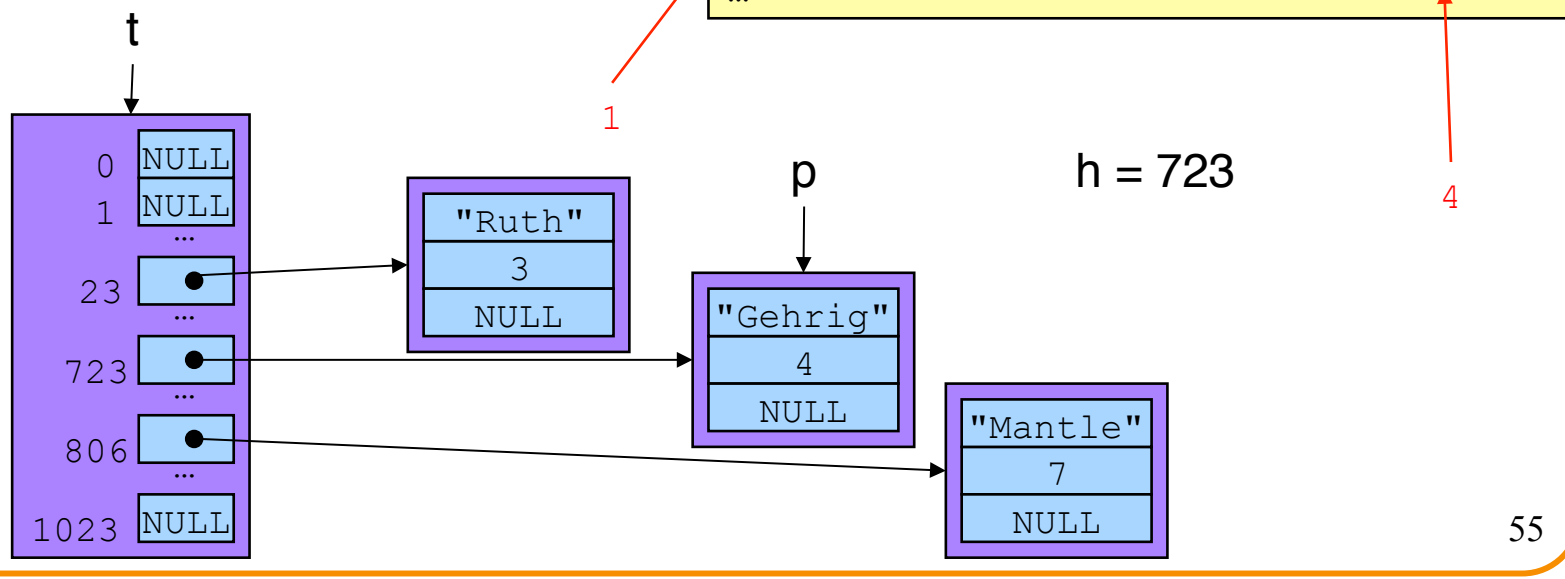
```
struct Table *t;
int value;
int found;
…
found =
    Table_search(t, "Gehrig", &value);
…
```



Hash Table: Search (5)

```
int Table_search(struct Table *t,
    const char *key, int *value) {
    struct Node *p;
    int h = hash(key);
    for (p = t->array[h]; p != NULL; p = p->next)
        if (strcmp(p->key, key) == 0) {
            *value = p->value;
            return 1;
        }
    return 0;
}
```

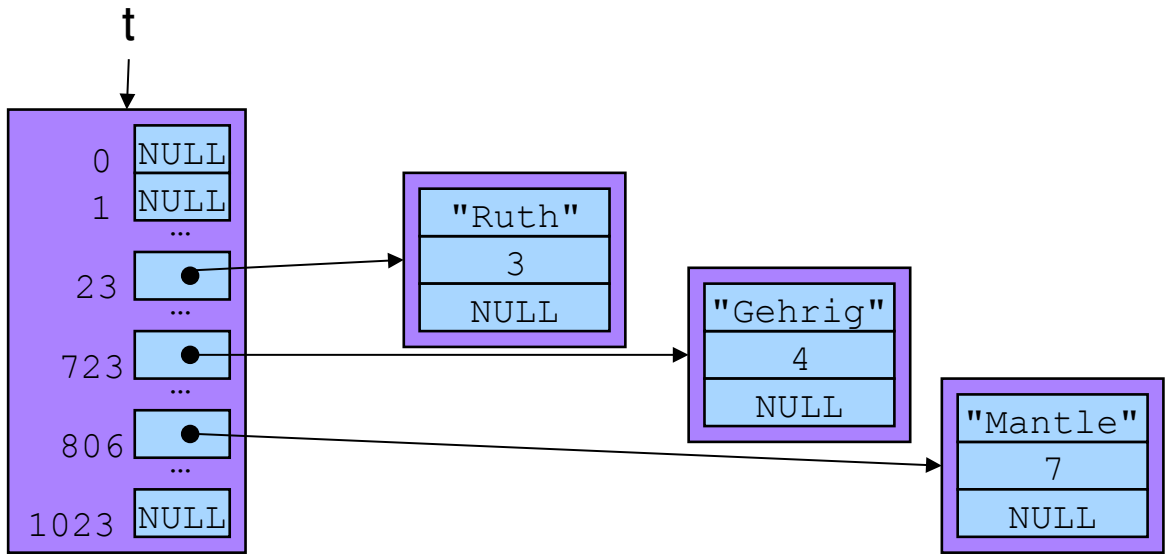
```
struct Table *t;
int value;
int found;
…
found =
    Table_search(t, "Gehrig", &value);
…
```



Hash Table: Free (1)

```
void Table_free(struct Table *t) {
    struct Node *p;
    struct Node *nextp;
    int b;
    for (b = 0; b < BUCKET_COUNT; b++)
        for (p = t->array[b]; p != NULL; p = nextp) {
            nextp = p->next;
            free(p);
        }
    free(t);
}
```

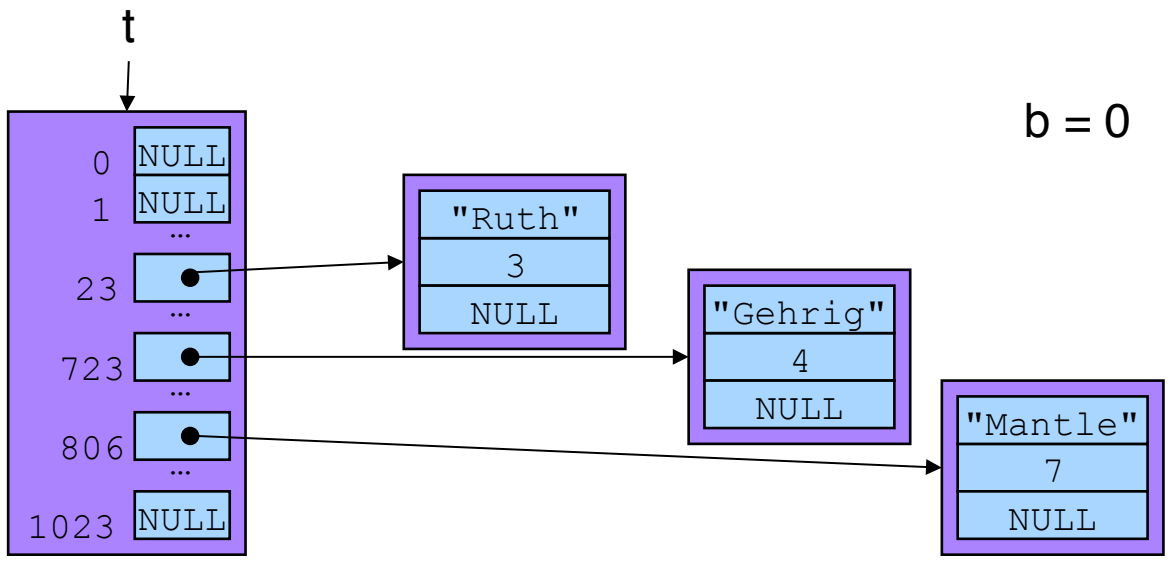
```
struct Table *t;
…
Table_free(t);
…
```



Hash Table: Free (2)

```
void Table_free(struct Table *t) {
   struct Node *p;
   struct Node *nextp;
   int b;
   for (b = 0; b < BUCKET_COUNT; b++)
      for (p = t->array[b]; p != NULL; p = nextp) {
         nextp = p->next;
         free(p);
      }
   free(t);
}
```

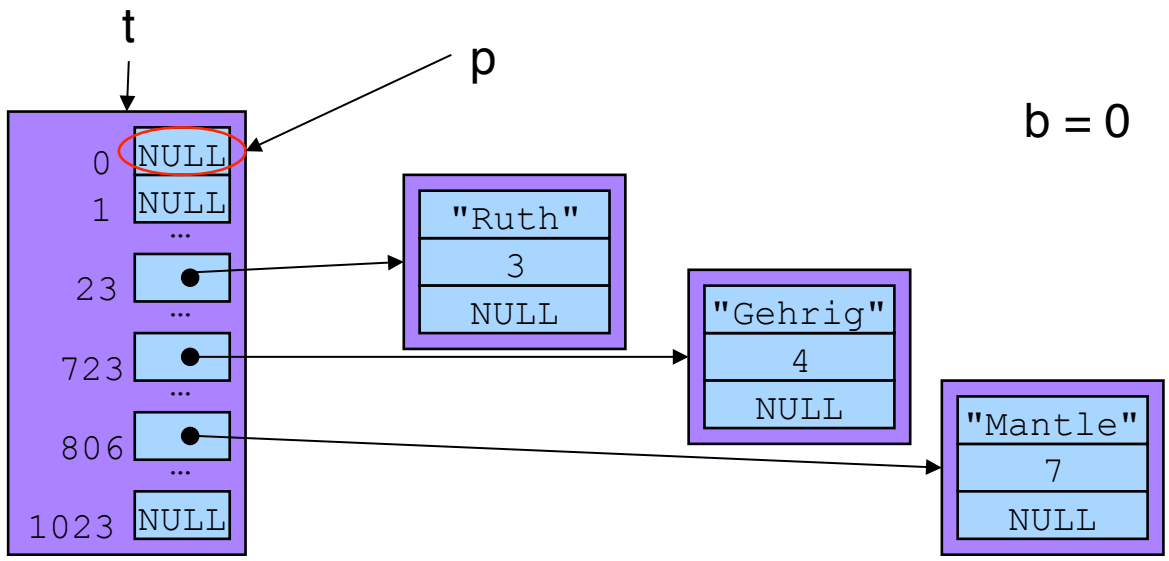
```
struct Table *t;
…
Table_free(t);
…
```



Hash Table: Free (3)

```
void Table_free(struct Table *t) {
    struct Node *p;
    struct Node *nextp;
    int b;
    for (b = 0; b < BUCKET_COUNT; b++)
        for (p = t->array[b]; p != NULL; p = nextp) {
            nextp = p->next;
            free(p);
        }
    free(t);
}
```

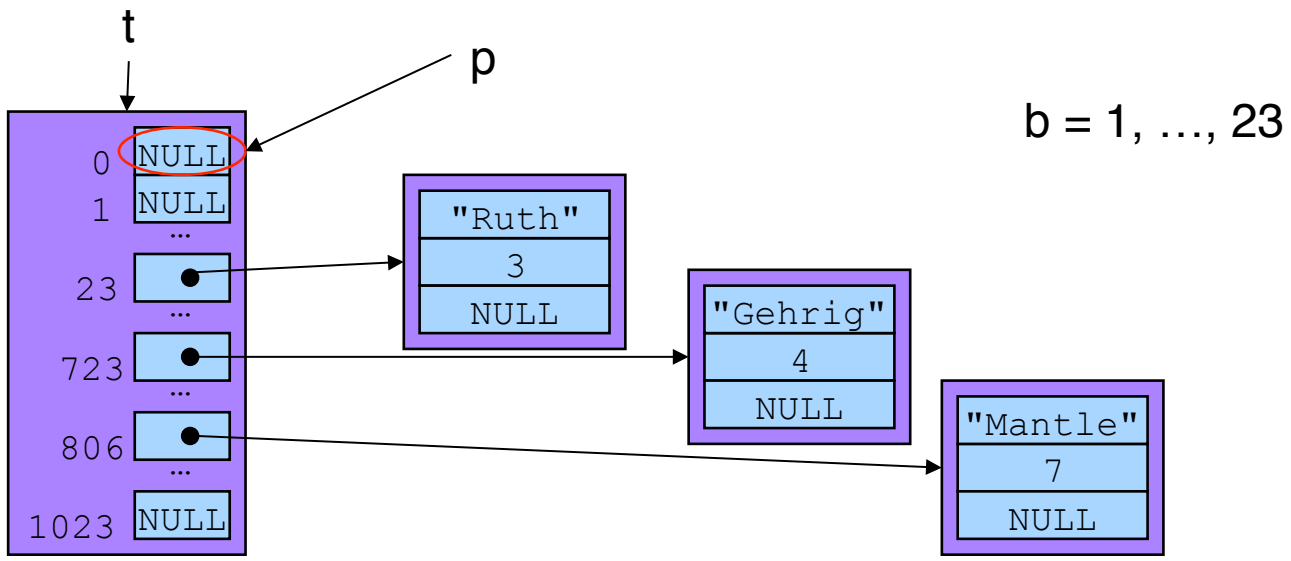
```
struct Table *t;
…
Table_free(t);
…
```



Hash Table: Free (4)

```
void Table_free(struct Table *t) {
    struct Node *p;
    struct Node *nextp;
    int b;
    for (b = 0; b < BUCKET_COUNT; b++)
        for (p = t->array[b]; p != NULL; p = nextp) {
            nextp = p->next;
            free(p);
        }
    free(t);
}
```

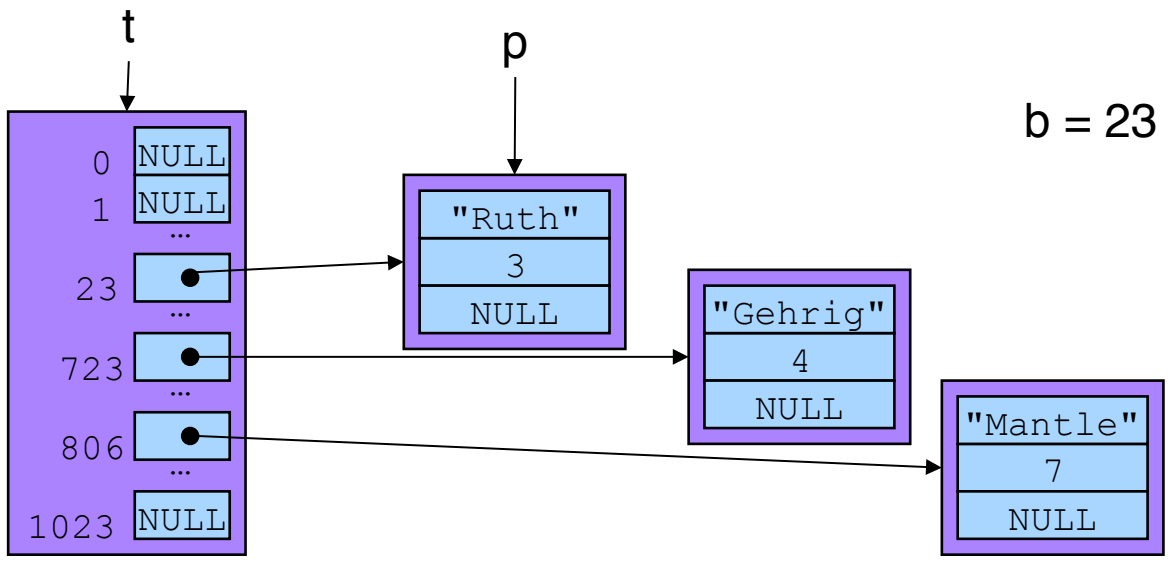
```
struct Table *t;
…
Table_free(t);
…
```



Hash Table: Free (5)

```
void Table_free(struct Table *t) {
    struct Node *p;
    struct Node *nextp;
    int b;
    for (b = 0; b < BUCKET_COUNT; b++)
        for (p = t->array[b]; p != NULL; p = nextp) {
            nextp = p->next;
            free(p);
        }
    free(t);
}
```

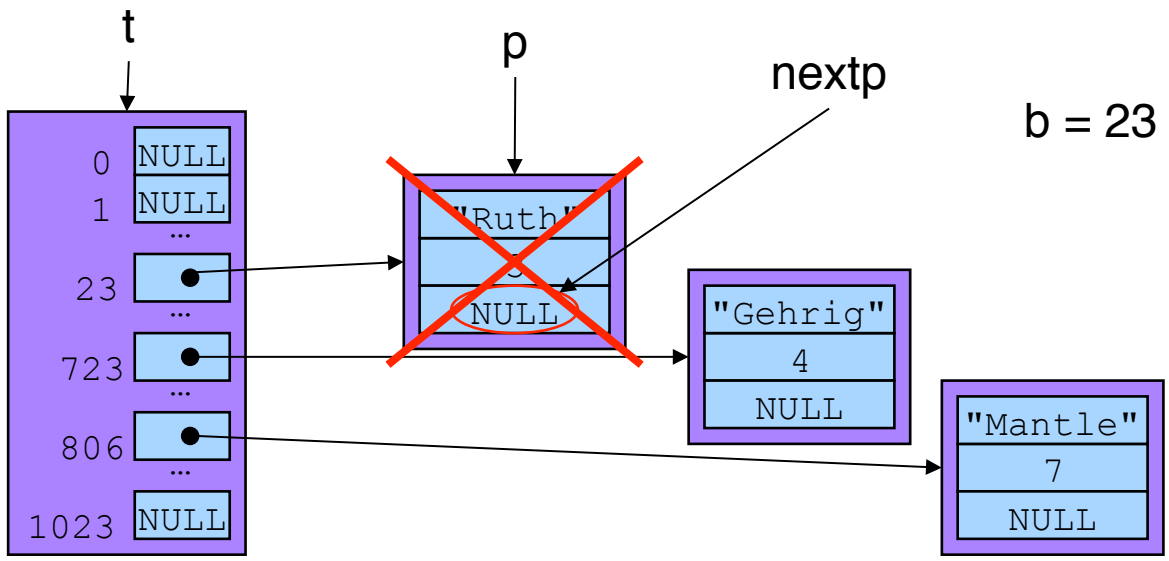
```
struct Table *t;
…
Table_free(t);
…
```



Hash Table: Free (6)

```
void Table_free(struct Table *t) {
    struct Node *p;
    struct Node *nextp;
    int b;
    for (b = 0; b < BUCKET_COUNT; b++)
        for (p = t->array[b]; p != NULL; p = nextp) {
            nextp = p->next;
            free(p);
        }
    free(t);
}
```

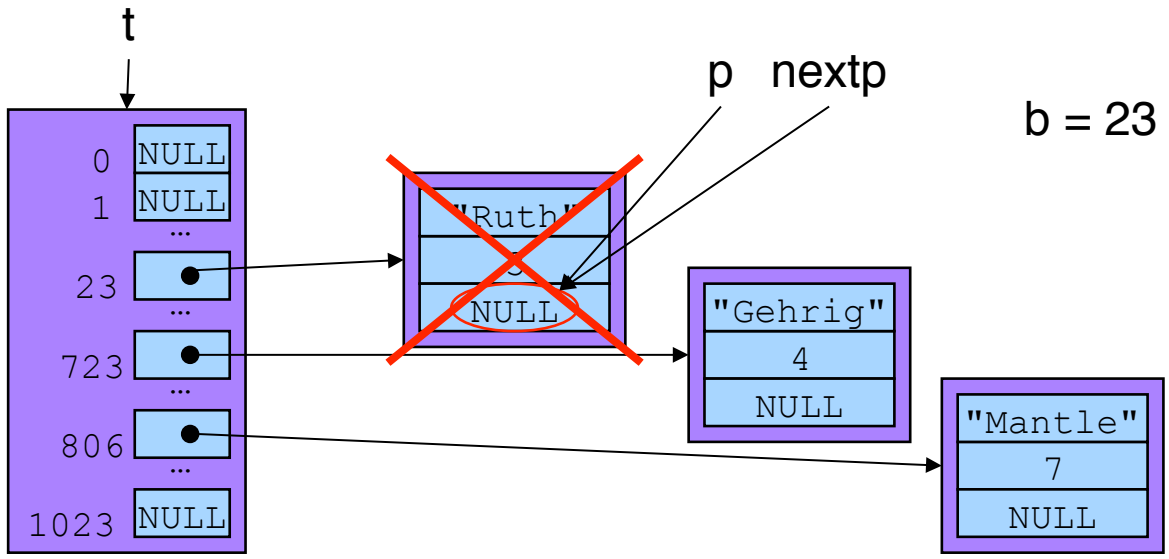
```
struct Table *t;
…
Table_free(t);
…
```



Hash Table: Free (7)

```
void Table_free(struct Table *t) {
    struct Node *p;
    struct Node *nextp;
    int b;
    for (b = 0; b < BUCKET_COUNT; b++)
        for (p = t->array[b]; p != NULL; p = nextp) {
            nextp = p->next;
            free(p);
        }
    free(t);
}
```

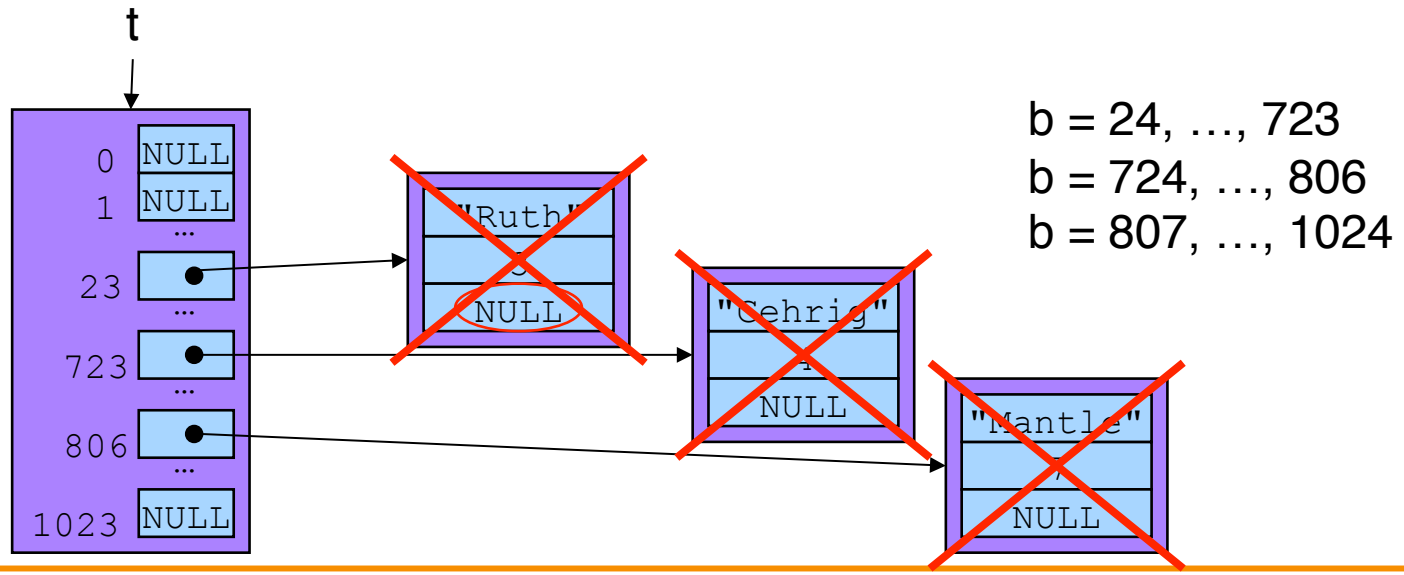
```
struct Table *t;
…
Table_free(t);
…
```



Hash Table: Free (8)

```
void Table_free(struct Table *t) {
    struct Node *p;
    struct Node *nextp;
    int b;
    for (b = 0; b < BUCKET_COUNT; b++)
        for (p = t->array[b]; p != NULL; p = nextp) {
            nextp = p->next;
            free(p);
        }
    free(t);
}
```

```
struct Table *t;
…
Table_free(t);
…
```

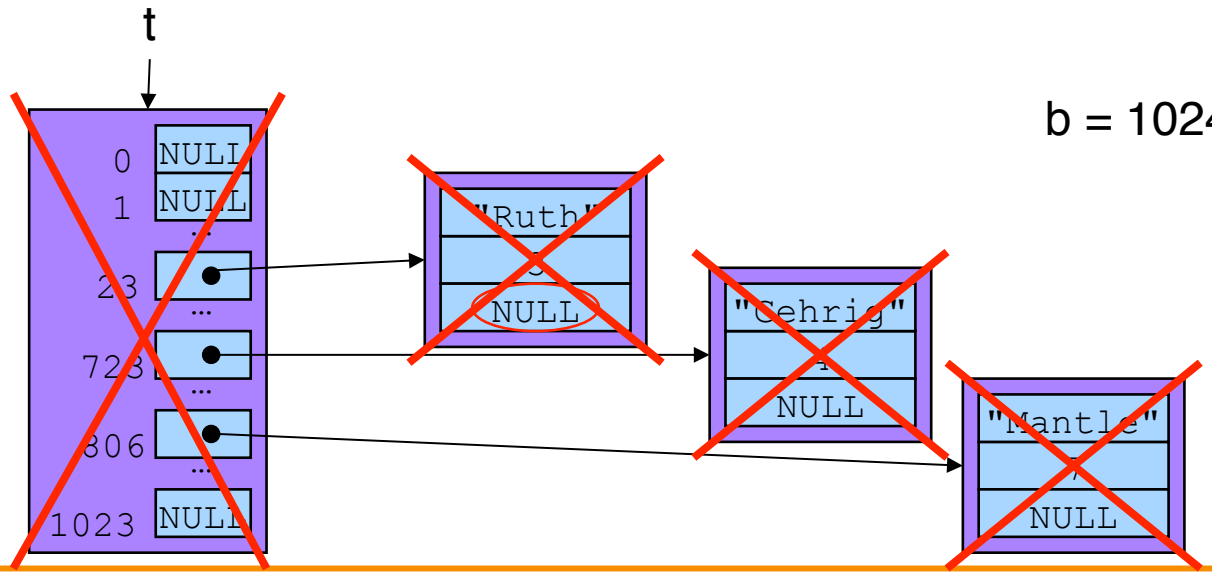


b = 24, …, 723
b = 724, …, 806
b = 807, …, 1024

Hash Table: Free (9)

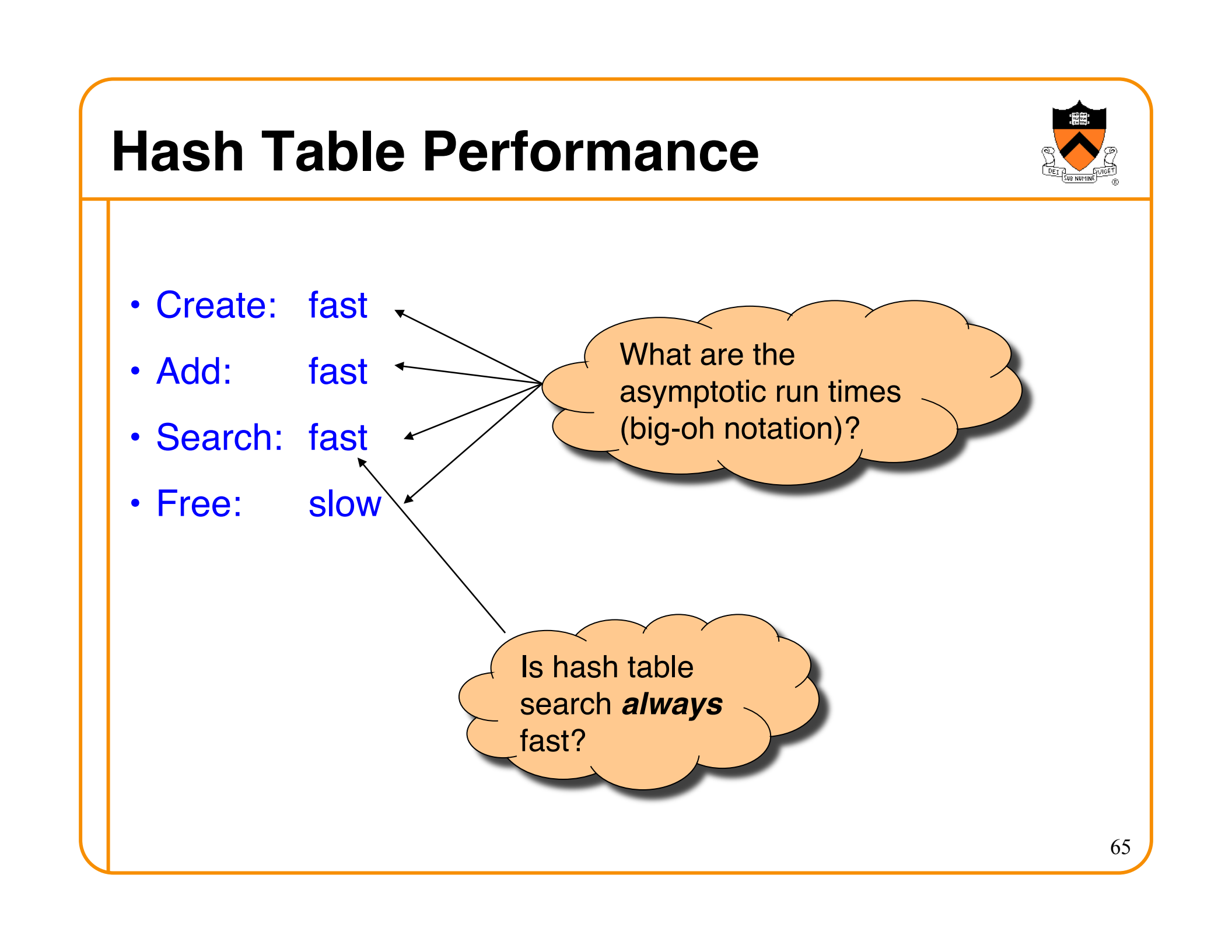
```
void Table_free(struct Table *t) {
    struct Node *p;
    struct Node *nextp;
    int b;
    for (b = 0; b < BUCKET_COUNT; b++)
        for (p = t->array[b]; p != NULL; p = nextp) {
            nextp = p->next;
            free(p);
        }
    free(t);
}
```

```
struct Table *t;
…
Table_free(t);
…
```



b = 1024

Hash Table Performance



- Create: fast
- Add: fast
- Search: fast
- Free: slow

What are the asymptotic run times (big-oh notation)?

Is hash table search ***always*** fast?

Key Ownership

- Note: Table_add() functions contain this code:

```
void Table_add(struct Table *t, const char *key, int value) {
    …
    struct Node *p = (struct Node*)malloc(sizeof(struct Node));
    p->key = key;
    …
}
```

  - Caller passes key, which is a pointer to memory where a string resides
  - Table_add() function stores within the table the address where the string resides

Key Ownership (cont.)

- Problem: Consider this calling code:

```
struct Table t;
char k[100] = "Ruth";
…
Table_add(t, k, 3);
strcpy(k, "Gehrig");
…
```

- Via Table_add(), table contains memory address k
- Client changes string at memory address k
- Thus client changes key within table

What happens if the client searches t for "Ruth"?

What happens if the client searches t for "Gehrig"?

Key Ownership (cont.)

- Solution: Table_add() saves **copy** of given key

```
void Table_add(struct Table *t, const char *key, int value) {
    …
    struct Node *p = (struct Node*)malloc(sizeof(struct Node));
    p->key = (const char*)malloc(strlen(key) + 1);
    strcpy(p->key, key);
    …
}
```

Why add 1?

  - If client changes string at memory address k, data structure is *not* affected

- Then the data structure "owns" the copy, that is:
  - The data structure is responsible for freeing the memory in which the copy resides
  - The Table_free() function must free the copy

Summary

- Common data structures & associated algorithms
  - Linked list
    - Fast insert, slow search
  - Hash table
    - Fast insert, (potentially) fast search
    - Invaluable for storing key/value pairs
    - Very common
- Related issues
  - Hashing algorithms
  - Memory ownership

Appendix

- “Stupid programmer tricks” related to hash tables…

Revisiting Hash Functions

- Potentially expensive to compute "mod c"
  - Involves division by c and keeping the remainder
  - Easier when c is a power of 2 (e.g., 16 = 2^4)
- An alternative (by example)
  - 53 = 32 + 16 + 4 + 1

• • •	32	16	8	4	2	1
0	0	1	1	0	1	0	1

  - 53 % 16 is 5, the last four bits of the number

• • •	32	16	8	4	2	1
0	0	0	0	0	1	0	1

  - Would like an easy way to isolate the last four bits…

Recall: Bitwise Operators in C

- Bitwise AND (&)

&	0	1
0	0	0
1	0	1

  - Mod on the cheap!
    - E.g., h = 53 & 15;

53	0	0	1	1	0	1	0	1
& 15	0	0	0	0	1	1	1	1
5	0	0	0	0	0	1	0	1

- Bitwise OR (|)

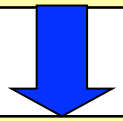
|	0	1
0	0	1
1	1	1

- One's complement (~)
  - Turns 0 to 1, and 1 to 0
  - E.g., set last three bits to 0
    - x = x & ~7;

A Faster Hash Function

```
unsigned int hash(const char *x) {
   int i;
   unsigned int h = 0U;
   for (i=0; x[i]!='\0'; i++)
      h = h * 65599 + (unsigned char)x[i];
   return h % 1024;
}
```

Previous version



```
unsigned int hash(const char *x) {
   int i;
   unsigned int h = 0U;
   for (i=0; x[i]!='\0'; i++)
      h = h * 65599 + (unsigned char)x[i];
   return h & 1023;
}
```

Faster

What happens if you mistakenly write “h & 1024”?

Speeding Up Key Comparisons

- Speeding up key comparisons
  - For any non-trivial value comparison function
  - Trick: store full hash result in structure

```
int Table_search(struct Table *t,
    const char *key, int *value) {
    struct Node *p;
    int h = hash(key); /* No % in hash function */
    for (p = t->array[h%1024]; p != NULL; p = p->next)
        if ((p->hash == h) && strcmp(p->key, key) == 0) {
            *value = p->value;
            return 1;
        }
    return 0;
}
```

Why is this so much faster?

Modularity

Jennifer Rexford

The material for this lecture is drawn, in part, from
The Practice of Programming (Kernighan & Pike) Chapter 4

Goals of this Lecture

- Help you learn:
  - How to create high quality modules in C
- Why?
  - Abstraction is a powerful (the only?) technique available for understanding large, complex systems
  - A power programmer knows how to find the abstractions in a large program
  - A power programmer knows how to convey a large program's abstractions via its modularity
- Seven module design heuristics...

Interfaces

(1) A well-designed module separates interface and implementation

- Why?
  - Hides implementation details from clients
    - Thus facilitating abstraction
  - Allows separate compilation of each implementation
    - Thus allowing partial builds

Interface Example 1

- Stack: A stack whose items are strings
  - Data structure
    - Linked list
  - Algorithms
    - **new**: Create a new Stack object and return it (or NULL if not enough memory)
    - **free**: Free the given Stack object
    - **push**: Push the given string onto the given Stack object and return 1 (or 0 if not enough memory)
    - **top**: Return the top item of the given Stack object
    - **pop**: Pop a string from the given Stack object and discard it
    - **isEmpty**: Return 1 the given Stack object is empty, 0 otherwise

Interfaces Example 1

- Stack (version 1)

```
/* stack.c */

struct Node {
   const char *item;
   struct Node *next;
};
struct Stack {
   struct Node *first;
};

struct Stack *Stack_new(void) {…}
void  Stack_free(struct Stack *s) {…}
int   Stack_push(struct Stack *s, const char *item) {…}
char *Stack_top(struct Stack *s) {…}
void  Stack_pop(struct Stack *s) {…}
int   Stack_isEmpty(struct Stack *s) {…}
```

```
/* client.c */

#include "stack.c"

/* Use the functions
defined in stack.c. */
```

- Stack module consists of one file (stack.c); no interface
- Problem: Change stack.c => must rebuild stack.c ***and client***
- Problem: Client "sees" Stack function definitions; poor abstraction

Interfaces Example 1

- Stack (version 2)

```
/* stack.h */

struct Node {
   const char *item;
   struct Node *next;
};
struct Stack {
   struct Node *first;
};

struct Stack *Stack_new(void);
void  Stack_free(struct Stack *s);
int   Stack_push(struct Stack *s, const char *item);
char *Stack_top(struct Stack *s);
void  Stack_pop(struct Stack *s);
int   Stack_isEmpty(struct Stack *s);
```

- Stack module consists of two files:

  (1) stack.h (the interface) declares functions and defines data structures

Interfaces Example 1

- Stack (version 2)

```
/* stack.c */

#include "stack.h"

struct Stack *Stack_new(void) {…}
void  Stack_free(struct Stack *s) {…}
int   Stack_push(struct Stack *s, const char *item) {…}
char *Stack_top(struct Stack *s) {…}
void  Stack_pop(struct Stack *s) {…}
int   Stack_isEmpty(struct Stack *s) {…}
```

(2) stack.c (the implementation) defines functions

- #includes stack.h so
  - Compiler can check consistency of function declarations and definitions
  - Functions have access to data structures

Interfaces Example 1

- Stack (version 2)

```
/* client.c */

#include "stack.h"

/* Use the functions declared in stack.h. */
```

  - Client #includes only the interface
  - Change stack.c => must rebuild stack.c, ***but not the client***
  - Client does not “see” Stack function definitions; better abstraction

Interface Example 2

- string (also recall Str from Assignment 2)

```
/* string.h */

size_t strlen(const char *s);
char  *strcpy(char *dest, const char *src);
char  *strncpy(char *dest, const char *src, size_t n);
char  *strcat(char *dest, const char *src);
char  *strncat(char *dest, const char *src, size_t n);
int   strcmp(const char *s, const char *t);
int   strncmp(const char *s, const char *t, size_t n);
char  *strstr(const char *haystack, const char *needle);
…
```

Interface Example 3

- stdio (from C90, vastly simplified)

```
/* stdio.h */

struct FILE {
   int cnt;     /* characters left */
   char *ptr;   /* next character position */
   char *base;  /* location of buffer */
   int flag;    /* mode of file access */
   int fd;      /* file descriptor */
};

#define OPEN_MAX 20
FILE _iob[OPEN_MAX];

#define stdin  (&_iob[0]);
#define stdout (&_iob[1]);
#define stderr (&_iob[2]);
...
```

Don't be concerned with details

Interface Example 3

- stdio (cont.)

```
…
FILE *fopen(const char *filename, const char *mode);
int   fclose(FILE *f);
int   fflush(FILE *f);

int   fgetc(FILE *f);
int   getc(FILE *f);
int   getchar(void);

int   putc(int c, FILE *f);
int   putchar(int c);

int   fscanf(FILE *f, const char *format, …);
int   scanf(const char *format, …);

int   fprintf(FILE *f, const char *format, …);
int   printf(const char *format, …);
…
```

Encapsulation

(2) A well-designed module encapsulates data

- An interface should hide implementation details
- A module should use its functions to encapsulate its data
- A module should not allow clients to manipulate the data directly

- Why?
  - **Clarity**: Encourages abstraction
  - **Security**: Clients cannot corrupt object by changing its data in unintended ways
  - **Flexibility**: Allows implementation to change – even the data structure – without affecting clients

Encapsulation Example 1

- Stack (version 1)

```
/* stack.h */

struct Node {
   const char *item;
   struct Node *next;
};
struct Stack {
   struct Node *first;
};

struct Stack *Stack_new(void);
void  Stack_free(struct Stack *s);
void  Stack_push(struct Stack *s, const char *item);
char *Stack_top(struct Stack *s);
void  Stack_pop(struct Stack *s);
int   Stack_isEmpty(struct Stack *s);
```

Structure type definitions in .h file

- That's bad
- Interface reveals how Stack object is implemented (e.g., as a linked list)
- Client can access/change data directly; could corrupt object

Encapsulation Example 1

- Stack (version 2)

Move definition of struct Node to implementation; clients need not know about it

Place ***declaration*** of struct Stack in interface; move ***definition*** to implementation

```
/* stack.h */

struct Stack;

struct Stack *Stack_new(void);
void  Stack_free(struct Stack *s);
void  Stack_push(struct Stack *s, const char *item);
char *Stack_top(struct Stack *s);
void  Stack_pop(struct Stack *s);
int   Stack_isEmpty(struct Stack *s);
```

- That's better
- Interface does not reveal how Stack object is implemented
- Client cannot access data directly

Encapsulation Example 1

- Stack (version 3)

```
/* stack.h */

typedef struct Stack * Stack_T;

Stack_T Stack_new(void);
void  Stack_free(Stack_T s);
void  Stack_push(Stack_T s, const char *item);
char *Stack_top(Stack_T s);
void  Stack_pop(Stack_T s);
int   Stack_isEmpty(Stack_T s);
```

Opaque pointer

- That's better still
- Interface provides "Stack_T" abbreviation for client
- Interface encourages client to view a Stack as an object, not as a (pointer to a) structure
- Client still cannot access data directly; data is "opaque" to the client

Encapsulation Example 2

- string
  - “Stateless” module
  - Has no state to encapsulate!

Encapsulation Example 3

- stdio

```
/* stdio.h */

struct FILE {
   int cnt;     /* characters left */
   char *ptr;   /* next character position */
   char *base;  /* location of buffer */
   int flag;    /* mode of file access */
   int fd;      /* file descriptor */
};
...
```

Structure type definition in .h file

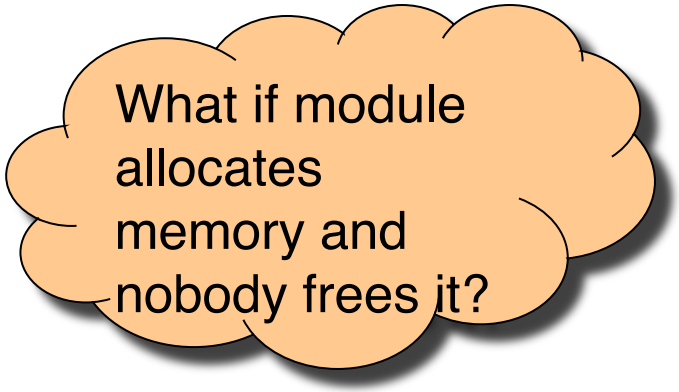
- Violates the heuristic
- Programmers can access data directly
  - Can corrupt the FILE object
  - Can write non-portable code
- But the functions are well documented, so
  - Few programmers examine stdio.h
  - Few programmers are tempted to access the data directly

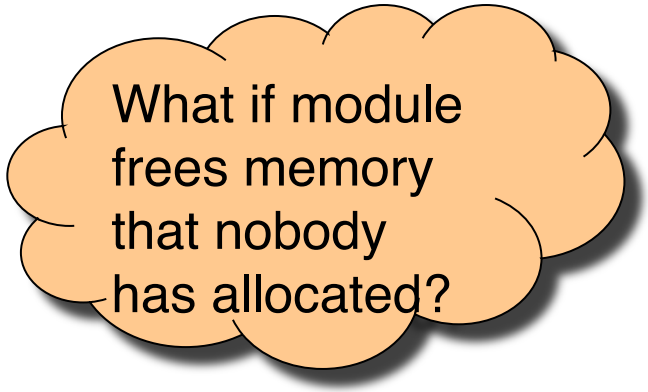
Resources

(3) A well-designed module manages resources consistently

- A module should free a resource if and only if the module has allocated that resource
- Examples
  - Object allocates memory <=> object frees memory
  - Object opens file <=> object closes file

- Why?
  - Error-prone to allocate and free resources at different levels





Resources Example 1

- Stack: Who allocates and frees the strings?
  - Reasonable options:

    (1) Client allocates and frees strings
    - **`Stack_push()`** does not create copy of given string
    - **`Stack_pop()`** does not free the popped string
    - **`Stack_free()`** does not free remaining strings

    (2) Stack object allocates and frees strings
    - **`Stack_push()`** creates copy of given string
    - **`Stack_pop()`** frees the popped string
    - **`Stack_free()`** frees all remaining strings

  - Our choice: (1)

Resources Examples 2, 3

- string
  - Stateless module
  - Has no resources to manage!
- stdio
  - **`fopen()`** allocates memory, uses file descriptor
  - **`fclose()`** frees memory, releases file descriptor

SymTable Aside

- Consider SymTable (from Assignment 3)…
- Who allocates and frees the key strings?
  - Reasonable options:
    - (1) Client allocates and frees strings
      - **`SymTable_put()`** does not create copy of given string
      - **`SymTable_remove()`** does not free the string
      - **`SymTable_free()`** does not free remaining strings
    - (2) SymTable object allocates and frees strings
      - **`SymTable_put()`** creates copy of given string
      - **`SymTable_remove()`** frees the string
      - **`SymTable_free()`** frees all remaining strings
  - Our choice: (2)

Advantages/ disadvantages (recall last lecture)?

Passing Resource Ownership

- Passing resource ownership
  - Should note violations of the heuristic in function comments

```
/* somefile.h */

…

void *f(void);
/* …
   This function allocates memory for
   the returned object.  You (the caller)
   own that memory, and so are responsible
   for freeing it when you no longer
   need it. */

…
```

Consistency

(4) A well-designed module is consistent

- A function's name should indicate its module
  - Facilitates maintenance programming; programmer can find functions more quickly
  - Reduces likelihood of name collisions (from different programmers, different software vendors, etc.)
- A module's functions should use a consistent parameter order
  - Facilitates writing client code

Consistency Examples

- Stack
  - (+) Each function name begins with “Stack_”
  - (+) First parameter identifies Stack object

- string
  - (+) Each function name begins with “str”
  - (+) Destination string parameter comes before source string parameter; mimics assignment

Consistency Examples (cont.)

- stdio

```
…
FILE *fopen(const char *filename, const char *mode);
int   fclose(FILE *f);
int   fflush(FILE *f);

int   fgetc(FILE *f);
int   getc(FILE *f);
int   getchar(void);

int   putc(int c, FILE *f);
int   putchar(int c);

int   fscanf(FILE *f, const char *format, …);
int   scanf(const char *format, …);

int   fprintf(FILE *f, const char *format, …);
int   printf(const char *format, …);
…
```

Are function names consistent?

Is parameter order consistent?

Minimization

(5) A well-designed module has a minimal interface

- Function declaration should be in a module's interface if and only if:
  - The function is **necessary** to make objects complete, or
  - The function is **convenient** for many clients

- Why?
  - More functions => higher learning costs, higher maintenance costs

Minimization Example 1

- Stack

```
/* stack.h */

typedef struct Stack *Stack_T ;

Stack_T Stack_new(void);
void  Stack_free(Stack_T s);
void  Stack_push(Stack_T s, const char *item);
char *Stack_top(Stack_T s);
void  Stack_pop(Stack_T s);
int   Stack_isEmpty(Stack_T s);
```

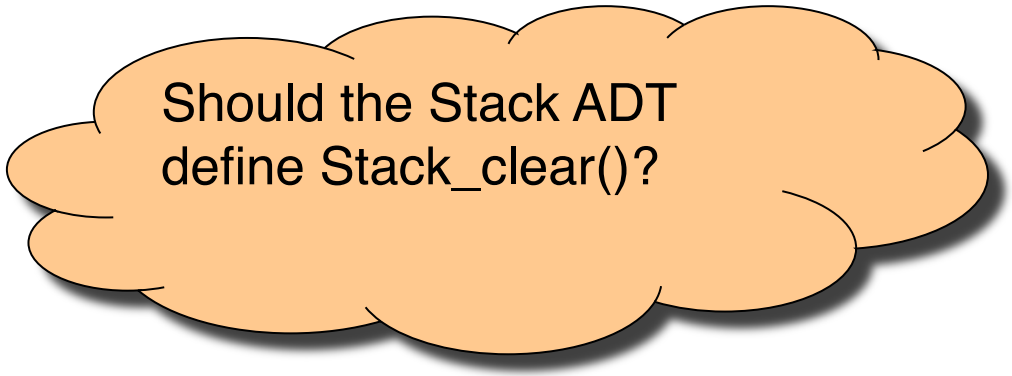
Should any functions be eliminated?

Minimization Example 1

- Another Stack function?

```
void Stack_clear(Stack_T s);
```

  - Pops all items from the Stack object



Minimization Example 2

- string

```
/* string.h */

size_t strlen(const char *s);
char  *strcpy(char *dest, const char *src);
char  *strncpy(char *dest, const char *src, size_t n);
char  *strcat(char *dest, const char *src);
char  *strncat(char *dest, const char *src, size_t n);
int    strcmp(const char *s, const char *t);
int    strncmp(const char *s, const char *t, size_t n);
char  *strstr(const char *haystack, const char *needle);
...
```

Should any functions be eliminated?

Minimization Example 3

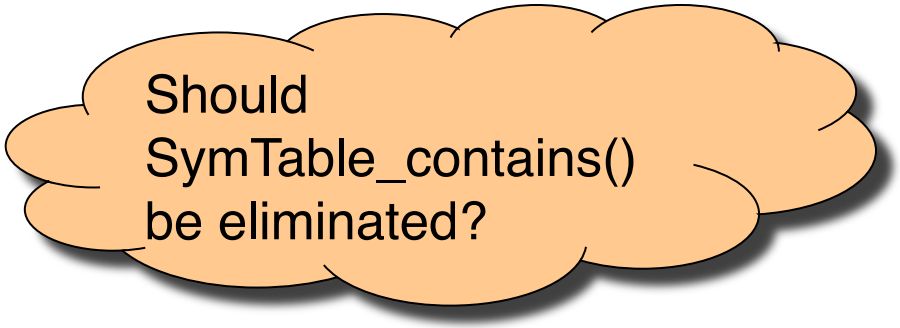
- stdio

```
…
FILE *fopen(const char *filename, const char *mode);
int   fclose(FILE *f);
int   fflush(FILE *f);

int   fgetc(FILE *f);
int   getc(FILE *f);
int   getchar(void);

int   putc(int c, FILE *f);
int   putchar(int c);

int   fscanf(FILE *f, const char *format, …);
int   scanf(const char *format, …);

int   fprintf(FILE *f, const char *format, …);
int   printf(const char *format, …);
…
```

Should any functions be eliminated?

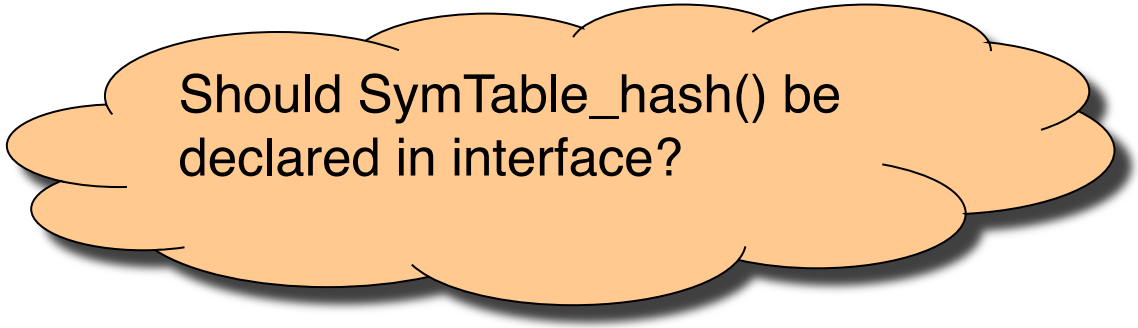
SymTable Aside

- Consider SymTable (from Assignment 3)
  - Declares **`SymTable_get()`** in interface
  - Declares **`SymTable_contains()`** in interface



SymTable Aside (cont.)

- Consider SymTable (from Assignment 3)
  - Defines **`SymTable_hash()`** in implementation



- Incidentally: In C any function should be either:
  - **Non-static**, and **declared** in the interface
  - **Static**, and **not declared** in the interface

Error Detection/Handling/Reporting

(6) A well-designed module detects and handles/ reports errors

- A module should:
  - **Detect** errors
  - **Handle** errors if it can; otherwise…
  - **Report** errors to its clients
    - A module often cannot assume what error-handling action its clients prefer

Detecting and Handling Errors in C

- C options for **detecting** errors
  - `if` statement
  - `assert` macro
- C options for **handling** errors
  - Print message to stderr
    - Impossible in many embedded applications
  - Recover and proceed
    - Sometimes impossible
  - Abort process
    - Often undesirable

Reporting Errors in C

- C options for **reporting** errors to client
  - Set **global variable**?
    - Easy for client to forget to check
    - Bad for multi-threaded programming
  - Use **function return value**?
    - Awkward if return value has some other natural purpose
  - Use extra **call-by-reference parameter**?
    - Awkward for client; must pass additional parameter
  - Call **`assert` macro**?
    - Terminates the entire program!
- No option is ideal

In contrast, Java supports "exceptions" (try-catch, throw)

User Errors

Our recommendation: Distinguish between…

(1) **User** errors

- Errors made by human user
- Errors that "could happen"

- Example: Bad data in stdin
- Example: Bad value of command-line argument

- Use `if` statement to detect
- Handle immediately if possible, or…
- Report to client via return value or call-by-reference parameter

Programmer Errors

(2) **Programmer** errors

- Errors made by a programmer
- Errors that "should never happen"

- Example: `int` parameter should not be negative, but is
- Example: pointer parameter should not be `NULL`, but is

- Use `assert` to detect and handle

- The distinction sometimes is unclear
  - Example: Write to file fails because disk is full

Error Handling Example 1

- Stack

```
/* stack.c */
…
int Stack_push(Stack_T s, const char *item) {
   struct Node *p;
   assert(s != NULL);
   p = (struct Node*)malloc(sizeof(struct Node));
   if (p == NULL) return 0;
   p->item = item;
   p->next = s->first;
   s->first = p;
   return 1;
}
```

- Invalid parameter is **programmer** error
  - Should never happen
  - Detect and handle via **`assert`**
- Memory allocation failure is **user** error
  - Could happen (huge data set and/or small computer)
  - Detect via **`if`**; report to client via return value

Error Handling Examples 2, 3

- string
  - No error detection or handling/reporting
  - Example: NULL parameter to **`strlen()`** => probable seg fault
- stdlib
  - Detects bad input
  - Uses function return values to report failure
    - Note awkwardness of **`scanf()`**
  - Sets global variable **`errno`** to indicate reason for failure

Establishing Contracts

(7) A well-designed module establishes contracts

- A module should establish contracts with its clients
- Contracts should describe what each function does, esp:
  - Meanings of parameters
  - Work performed
  - Meaning of return value
  - Side effects

- Why?
  - Facilitates cooperation between multiple programmers
  - Assigns blame to contract violators!!!
    - If your functions have precise contracts and implement them correctly, then the bug must be in someone else's code!!!

Establishing Contracts in C

- Our recommendation…
- In C, establish contracts via comments in module interface

Establishing Contracts Example

- Stack

```
/* stack.h */
…
int Stack_push(Stack_T s, const char *item);
/* Push item onto s.  Return 1 (TRUE)
   if successful, or 0 (FALSE) if
   insufficient memory is available. */

…
```

  - Comment defines contract:
    - Meaning of function's parameters
      - s is the stack to be affected; item is the item to be pushed
    - Work performed
      - Push item onto s
    - Meaning of return value
      - Indicates success/failure
    - Side effects
      - (None, by default)

Summary: Well-Designed Modules

- (1) Separates interface and implementation
- (2) Encapsulates data
- (3) Manages resources consistently
- (4) Is consistent
- (5) Has a minimal interface
- (6) Detects and handles/reports errors
- (7) Establishes contracts

Appendix

Two additional heuristics

which are more advanced in nature…

Strong Cohesion

(8) A well-designed module has strong cohesion

- A module's functions should be strongly related to each other

- Why?
  - Strong cohesion facilitates abstraction

Strong Cohesion Examples

- Stack
  - (+) All functions are related to the encapsulated data
- string
  - (+) Most functions are related to string handling
  - (-) Some functions are not related to string handling

    `memcpy()`, **`memmove()`**, **`memcmp()`**, **`memchr()`**, **`memset()`**
  - (+) But those functions are similar to string-handling functions
- stdio
  - (+) Most functions are related to I/O
  - (-) Some functions don't do I/O

    `sprintf()`, **`sscanf()`**
  - (+) But those functions are similar to I/O functions

Weak Coupling

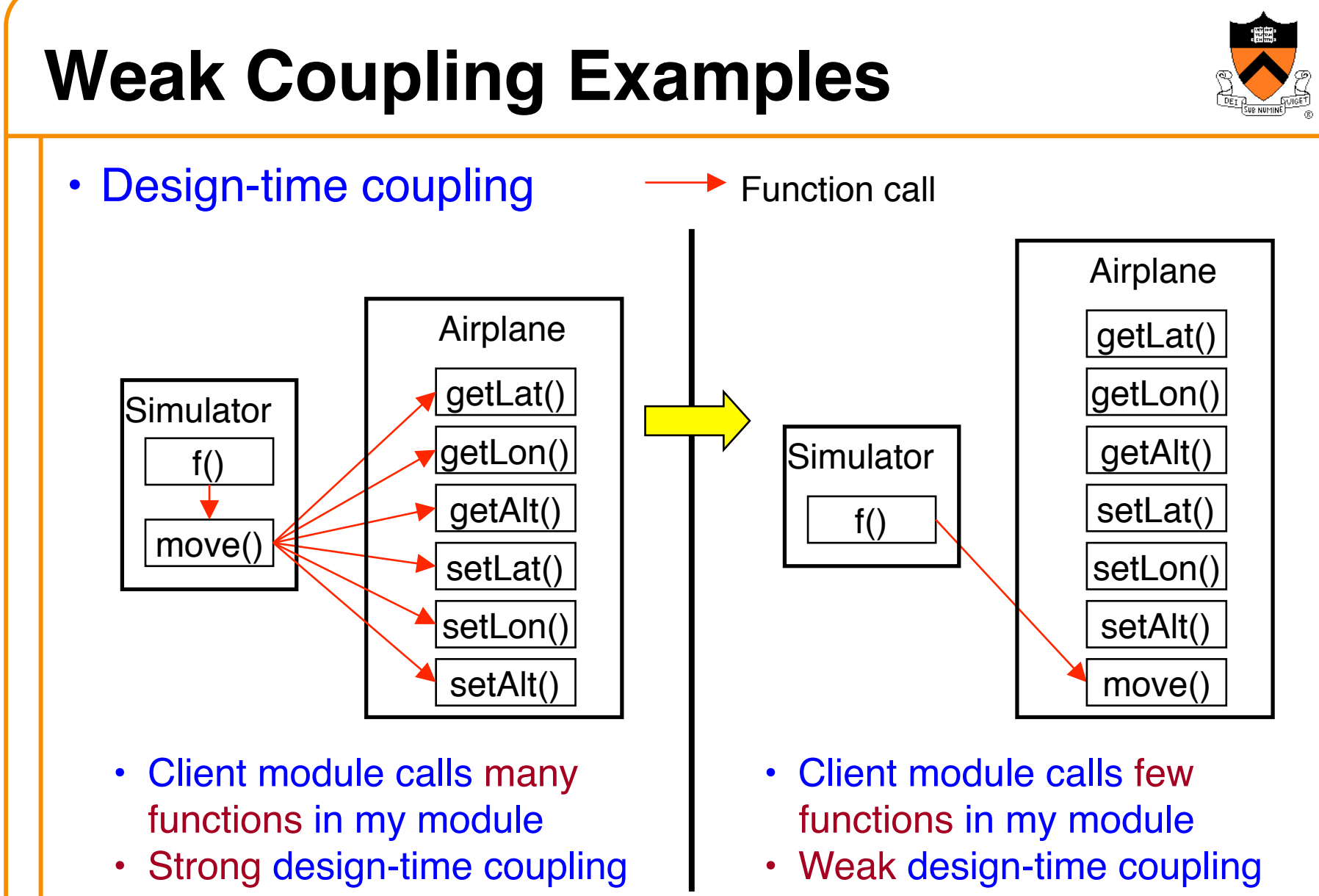
(9) A well-designed module has weak coupling

- Module should be weakly connected to other modules in program
- Interaction within modules should be more intense than interaction among modules

- Why? Theoretical observations
  - Maintenance: Weak coupling makes program easier to modify
  - Reuse: Weak coupling facilitates reuse of modules

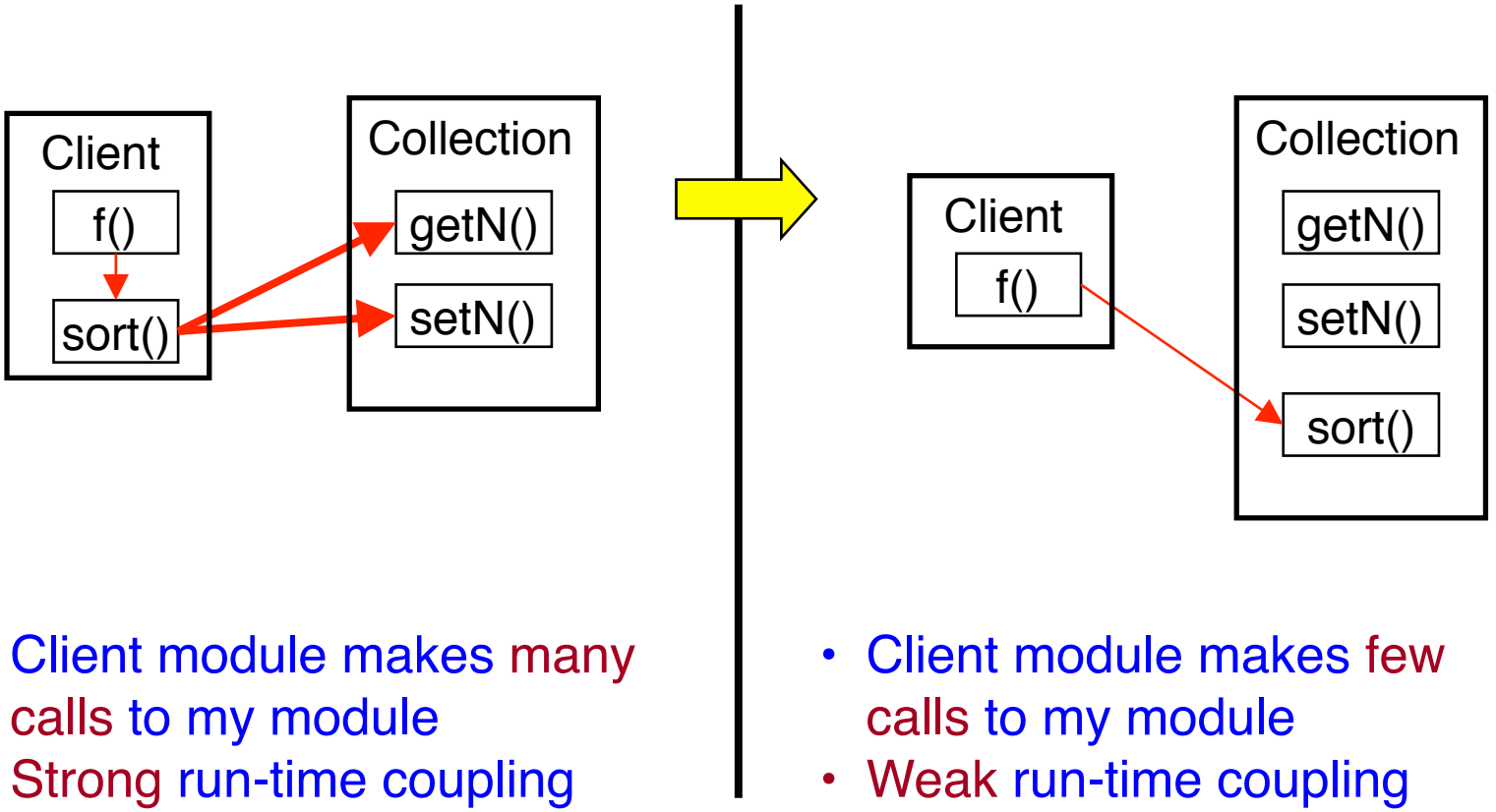
- Why? Empirical evidence
  - Empirically, modules that are weakly coupled have fewer bugs

Weak Coupling Examples



Weak Coupling Examples (cont.)

- Run-time coupling → Many function calls → One function call

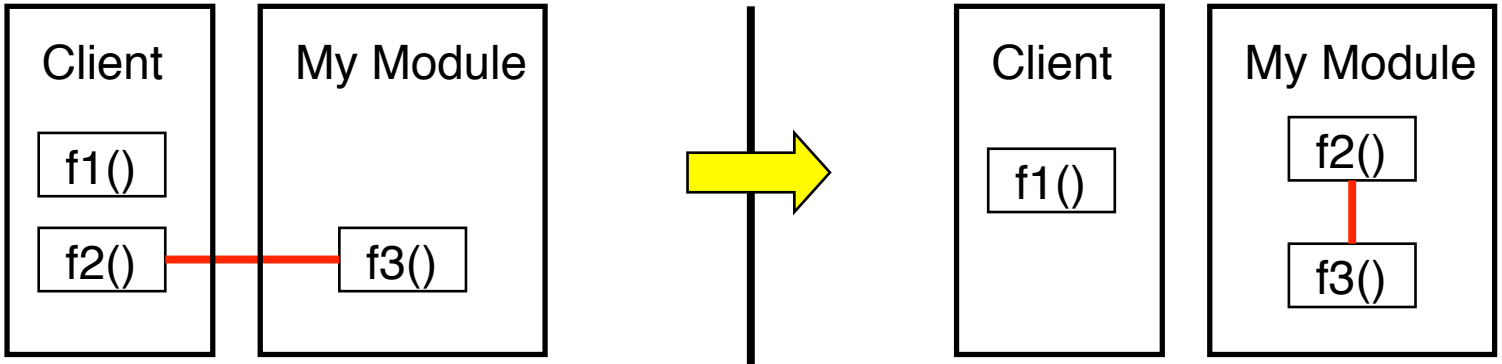


- Client module makes many calls to my module
- Strong run-time coupling

- Client module makes few calls to my module
- Weak run-time coupling

Weak Coupling Examples (cont.)

- Maintenance-time coupling



- Maintenance programmer changes client and my module together frequently
- Strong maintenance-time coupling

- Maintenance programmer changes client and my module together infrequently
- Weak maintenance-time coupling

Achieving Weak Coupling

- Achieving weak coupling could involve moving code:
  - From clients to my module (shown)
  - From my module to clients (not shown)
  - From clients and my module to a new module (not shown)

Summary

- A well-designed module:
  - (1) Separates interface and implementation
  - (2) Encapsulates data
  - (3) Manages resources consistently
  - (4) Is consistent
  - (5) Has a minimal interface
  - (6) Detects and handles/reports errors
  - (7) Establishes contracts
  - **(8) Has strong cohesion**
  - **(9) Has weak coupling**

Generics

Jennifer Rexford

Goals of this Lecture

- Generic modules
  - Data structures that can store multiple types of data
  - Functions that can work on multiple types of data
- How to create generic modules in C
  - Which wasn't designed with generic modules in mind!
- Why?
  - Reusing code is cheaper than writing new code
  - Generic modules are more reusable

Generic Data Structures Example

- Recall Stack module from last lecture

```
/* stack.h */

typedef struct Stack *Stack_T;

Stack_T Stack_new(void);
void  Stack_free(Stack_T s);
int   Stack_push(Stack_T s, const char *item);
char *Stack_top(Stack_T s);
void  Stack_pop(Stack_T s);
int   Stack_isEmpty(Stack_T s);
```

- Items are strings (type char*)

Generic Data Structures Example

- Stack operations (push, pop, top, etc.) make sense for items *other than* strings too
- So Stack module could (and maybe should) be generic
- Problem: How to make Stack module generic?

Generic Data Structures via typedef

- Solution 1: Let clients define item type

```
/* client.c */

struct Item {
   char *str; /* Or whatever is appropriate */
};

…
Stack_T s;
struct Item item;

item.str = "hello";
s = Stack_new();
Stack_push(s, item);
…
```

Do you see any problems with this approach?

```
/* stack.h */

typedef struct Item *Item_T;
typedef struct Stack *Stack_T;

Stack_T Stack_new(void);
void    Stack_free(Stack_T s);
int     Stack_push(Stack_T s, Item_T item);
Item_T  Stack_top(Stack_T s);
void    Stack_pop(Stack_T s);
int     Stack_isEmpty(Stack_T s);
```

Problems with "typedef" Approach

- Client must define structure type
  - …and create structures of that type
- Client might already use "Item_T" for some other purpose!
- Client might need two Stack objects
  - …holding different types of data!!!

Generic Data Structures via void*

- Solution 2: The generic pointer (void*)

```
/* stack.h */

typedef struct Stack *Stack_T;

Stack_T Stack_new(void);
void    Stack_free(Stack_T s);
int     Stack_push(Stack_T s, const void *item);
void   *Stack_top(Stack_T s);
void    Stack_pop(Stack_T s);
int     Stack_isEmpty(Stack_T s);
```

Generic Data Structures via void*

- Can assign a pointer of any type to a void pointer

```
/* client.c */

…
Stack_T s;
s = Stack_new();
Stack_push(s, "hello");
…
```

OK to match an actual parameter of type char* with a formal parameter of type void*

```
/* stack.h */

typedef struct Stack *Stack_T;

Stack_T Stack_new(void);
void    Stack_free(Stack_T s);
int     Stack_push(Stack_T s, const void *item);
void   *Stack_top(Stack_T s);
void    Stack_pop(Stack_T s);
int     Stack_isEmpty(Stack_T s);
```

Generic Data Structures via void*

- Can assign a void pointer to a pointer of any type

```
/* client.c */

char *str;
…
Stack_T s;
s = Stack_new();
Stack_push(s, "hello");
…
str = Stack_top(s);
```

OK to assign
a void* return value
to a char*

```
/* stack.h */

typedef struct Stack *Stack_T;

Stack_T Stack_new(void);
void    Stack_free(Stack_T s);
int     Stack_push(Stack_T s, const void *item);
void    *Stack_top(Stack_T s);
void    Stack_pop(Stack_T s);
int     Stack_isEmpty(Stack_T s);
```

Generic Data Structures via void*

- Problem: Client must know what type of data a void pointer is pointing to

```
/* client.c */

int *i;
…
Stack_T s;
s = Stack_new();
Stack_push(s, "hello");
…
i = Stack_top(s);
```

Client pushes a string

Client considers retrieved value to be a pointer to an int! Legal!!! Trouble!!!

- Solution: None
  - Void pointers subvert the compiler's type checking

Generic Data Structures via void*

- Problem: Stack items must be pointers
  - E.g. Stack items cannot be of primitive types (int, double, etc.)

```
/* client.c */
…
int i = 5;
…
Stack_T s;
s = Stack_new();
…
Stack_push(s, 5);
…
Stack_push(s, &i);
```

Not OK to match an actual parameter of type int with a formal parameter of type void*

OK, but awkward

- Solution in C: none
  - In C++: template classes and template functions
  - In Java: generic classes

Generic Algorithms Example

- Suppose we add another function to the Stack module

```
/* stack.h */

typedef struct Stack *Stack_T;

Stack_T Stack_new(void);
void    Stack_free(Stack_T s);
int     Stack_push(Stack_T s, const void *item);
void   *Stack_top(Stack_T s);
void    Stack_pop(Stack_T s);
int     Stack_isEmpty(Stack_T s);
int     Stack_areEqual(Stack_T s1, Stack_T s2);
```

Should return 1 (TRUE) iff s1 and s2 are equal, that is, contain equal items in the same order

Generic Algorithm Attempt 1

- Attempt 1

```
/* stack.c */

…

int Stack_areEqual(Stack_T s1, Stack_T s2) {
   return s1 == s2;
}
```

```
/* client.c */

char str1[] = "hi";
char str2[] = "hi";
Stack_T s1 = Stack_new();
Stack_T s2 = Stack_new();
Stack_push(s1, str1);
Stack_push(s2, str2);

if (Stack_areEqual(s1,s2)) {
   …
}
```

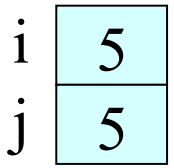
- Checks if s1 and s2 are **identical**, not **equal**
  - Compares pointers, not items
- That's not what we want

What does this return?

Addresses vs. Values

- Suppose two locations in memory have the same value

```
int i=5;
int j=5;
```



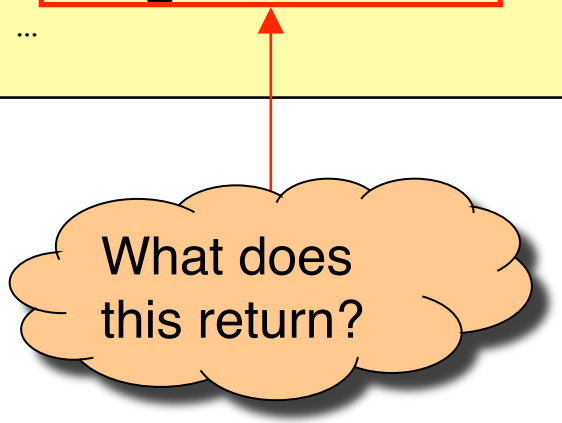
- The addresses of the variables are *not* the same
  - That is "(&i == &j)" is FALSE
- Need to compare the values themselves
  - That is "(i == j)" is TRUE
- Unfortunately, comparison operation is type specific
  - The "==" works for integers and floating-point numbers
  - But not for strings and more complex data structures

Generic Algorithm Attempt 2

- Attempt 2

```
/* stack.c */
…
int Stack_areEqual(Stack_T s1, Stack_T s2) {
   struct Node *p1 = s1->first;
   struct Node *p2 = s2->first;
   while ((p1 != NULL) && (p2 != NULL)) {
      if (p1 != p2)
         return 0;
      p1 = p1->next;
      p2 = p2->next;
   }
   if ((p1 != NULL) || (p2 != NULL))
      return 0;
   return 1;
}
```

```
/* client.c */

char str1[] = "hi";
char str2[] = "hi";
Stack_T s1 = Stack_new();
Stack_T s2 = Stack_new();
Stack_push(s1, str1);
Stack_push(s2, str2);

if (Stack_areEqual(s1,s2)) {
   …
}
```



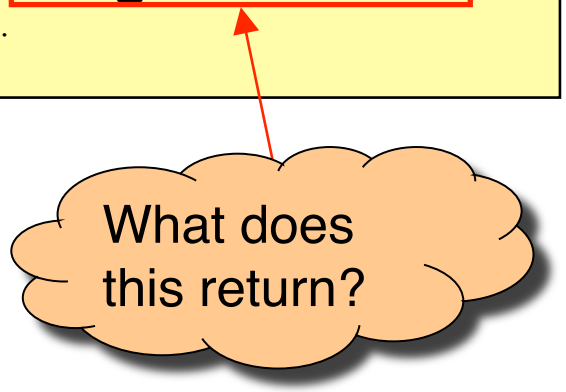
- Checks if **nodes** are **identical**
  - Compares pointers, not items
- That is *still* not what we want

Generic Algorithm Attempt 3

- Attempt 3

```
/* stack.c */
…
int Stack_areEqual(Stack_T s1, Stack_T s2) {
   struct Node *p1 = s1->first;
   struct Node *p2 = s2->first;
   while ((p1 != NULL) && (p2 != NULL)) {
      if (p1->item != p2->item)
         return 0;
      p1 = p1->next;
      p2 = p2->next;
   }
   if ((p1 != NULL) || (p2 != NULL))
      return 0;
   return 1;
}
```

```
/* client.c */

char str1[] = "hi";
char str2[] = "hi";
Stack_T s1 = Stack_new();
Stack_T s2 = Stack_new();
Stack_push(s1, str1);
Stack_push(s2, str2);

if (Stack_areEqual(s1,s2)) {
   …
}
```



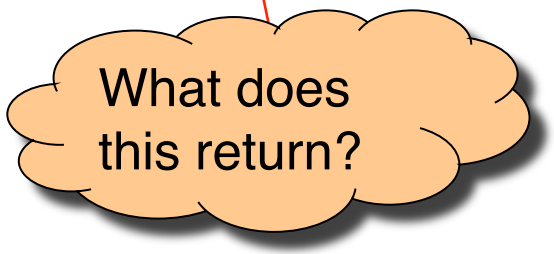
- Checks if **items** are **identical**
  - Compares pointers to items, not items themselves
- That is *still* not what we want

Generic Algorithm Attempt 4

- Attempt 4

```
/* stack.c */
…
int Stack_areEqual(Stack_T s1, Stack_T s2) {
   struct Node *p1 = s1->first;
   struct Node *p2 = s2->first;
   while ((p1 != NULL) && (p2 != NULL)) {
      if (strcmp(p1->item, p2->item) != 0)
         return 0;
      p1 = p1->next;
      p2 = p2->next;
   }
   if ((p1 != NULL) || (p2 != NULL))
      return 0;
   return 1;
}
```

```
/* client.c */

char str1[] = "hi";
char str2[] = "hi";
Stack_T s1 = Stack_new();
Stack_T s2 = Stack_new();
Stack_push(s1, str1);
Stack_push(s2, str2);

if (Stack_areEqual(s1,s2)) {
   …
}
```



- Checks if **items** are **equal**
- That's what we want
- But `strcmp()` works only if items are strings!
- How to compare values when we don't know their type?

Generic Algorithm via Function Pointer

- Attempt 5

```
/* stack.h */

typedef struct Stack *Stack_T;

Stack_T Stack_new(void);
void    Stack_free(Stack_T s);
int     Stack_push(Stack_T s, const void *item);
void   *Stack_top(Stack_T s);
void    Stack_pop(Stack_T s);
int     Stack_isEmpty(Stack_T s);
int     Stack_areEqual(Stack_T s1, Stack_T s2,
           int (*cmp)(const void *item1, const void *item2));
```

- Add parameter to **`Stack_areEqual()`**
  - Pointer to a compare function
- Allows client to supply the function that **`Stack_areEqual()`** should call to compare items

Generic Algorithm via Function Pointer

- Attempt 5 (cont.)

```
/* stack.c */
…
int Stack_areEqual(Stack_T s1, Stack_T s2,
   int (*cmp)(const void *item1, const void *item2)) {
   struct Node *p1 = s1->first;
   struct Node *p2 = s2->first;
   while ((p1 != NULL) && (p2 != NULL)) {
      if ((*cmp)(p1->item, p2->item) != 0)
         return 0;
      p1 = p1->next;
      p2 = p2->next;
   }
   if ((p1 != NULL) || (p2 != NULL))
      return 0;
   return 1;
}
```

- Definition of **`Stack_areEqual()`** uses the function pointer to call the client-supplied compare function
- **`Stack_areEqual()`** "calls back" into client code

Generic Algorithm via Function Pointer

- Attempt 5 (cont.)

```
/* client.c */

int strCompare(const void *item1, const void *item2) {
   char *str1 = item1;
   char *str2 = item2;
   return strcmp(str1, str2);
}
…
char str2[] = "hi";
Stack_T s1 = Stack_new();
Stack_T s2 = Stack_new();
Stack_push(s1, str1);
Stack_push(s2, str2);

if (Stack_areEqual(s1,s2,strCompare)) {
   …
}
```

Client passes address of **`strCompare()`** to **`Stack_areEqual()`**

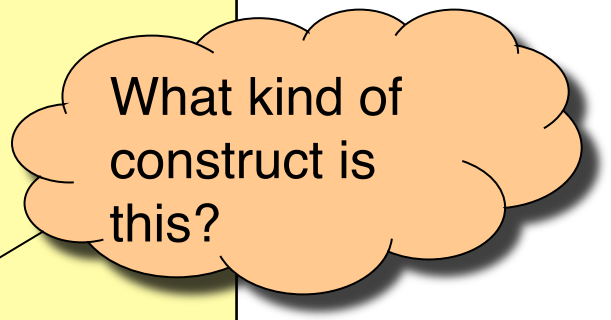
What does this return?

- Client defines "callback function", and passes pointer to it to **`Stack_areEqual()`**
- Callback function must match **`Stack_areEqual()`** parameter exactly

Generic Algorithm via Function Pointer

- Alternative: Client defines more "natural" callback function
- Attempt 5 (cont.)

```
/* client.c */

int strCompare(const char *str1, const char *str2) {
   return strcmp(str1, str2);
}
…
char str2[] = "hi";
Stack_T s1 = Stack_new();
Stack_T s2 = Stack_new();
Stack_push(s1, str1);
Stack_push(s2, str2);

if (Stack_areEqual(s1, s2,
   (int (*)(const void*, const void*))strCompare)) {
   …
}
```



Generic Algorithm via Function Pointer

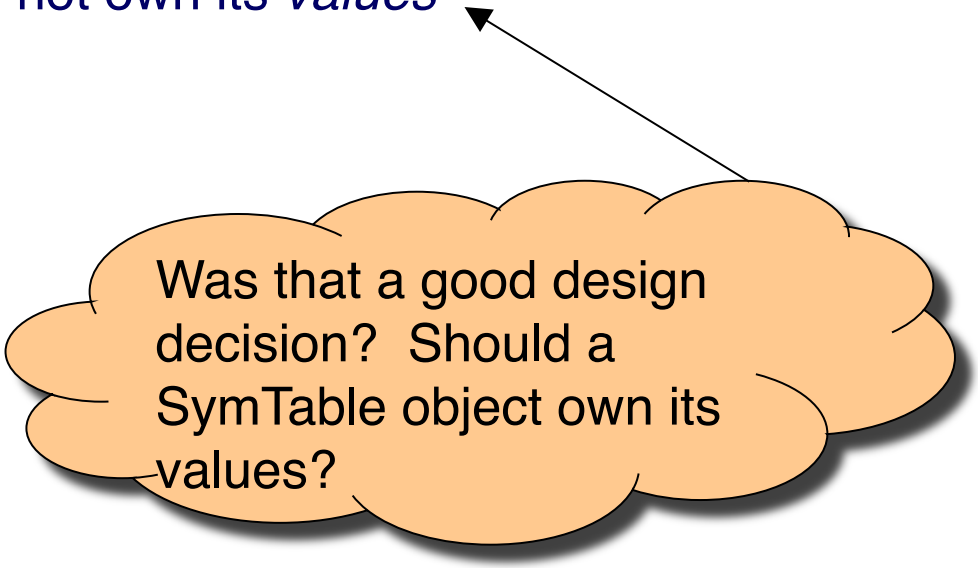
- Attempt 5 (cont.)

```
/* client.c */
…
char str2[] = "hi";
Stack_T s1 = Stack_new();
Stack_T s2 = Stack_new();
Stack_push(s1, str1);
Stack_push(s2, str2);

if (Stack_areEqual(s1, s2,
    (int (*)(const void*, const void*))strcmp)) {
    …
}
```

- Alternative (for string comparisons only): Simply use **`strcmp()`**!

SymTable Aside

- Consider SymTable (from Assignment 3)…
  - A SymTable object owns its *keys*
  - A SymTable object does not own its *values*



Summary

- Generic data structures
  - Via item typedef
    - Safe, but not realistic
  - Via the generic pointer (void*)
    - Limiting: items must be pointers
    - Dangerous: subverts compiler type checking
    - The best we can do in C
- Generic algorithms
  - Via function pointers and callback functions

Appendix: Wrappers

- Q: Can we make "void pointer" generic ADTs safer?
- A: Yes, with some extra work…

- Example: Suppose
  - We have a generic **Stack** ADT
    - Items are void pointers
  - We wish to create a **StrStack** ADT
    - Same as **Stack**, except items are strings (char pointers)

Appendix: Wrapper Interface

- Define type-specific interface

```
/* strstack.h */
…
typedef struct StrStack *StrStack_T;

StrStack_T StrStack_new(void);
void       StrStack_free(StrStack_T ss);
int        StrStack_push(StrStack_T ss, const char *item);
char      *StrStack_top(StrStack_T ss);
void       StrStack_pop(StrStack_T ss);
int        StrStack_isEmpty(StrStack_T ss);
…
```

Appendix: Wrapper Data Structure

- Define StrStack structure such that it has one field of type Stack_T

```
/* strstack.c */

struct StrStack {
   Stack_T s;
};

…
```

Appendix: Wrapper Functions

- Define StrStack_new() to call Stack_new()

```
/* strstack.c */
…
StrStack_T StrStack_new(void) {
   Stack_T s;
   StrStack_T ss;
   s = Stack_new();
   if (s == NULL)
      return NULL;
   ss = (StrStack_T)malloc(sizeof(struct StrStack));
   if (ss == NULL) {
      Stack_free(s);
      return NULL;
   }
   ss->s = s;
   return ss;
}
…
```

Appendix: Wrapper Functions

- Define StrStack_free() to call Stack_free()

```
/* strstack.c */
…
void StrStack_free(StrStack_T ss) {
   Stack_free(ss->s);
   free(ss);
}
…
```

Appendix: Wrapper Functions

- Define remaining StrStack functions to call corresponding Stack functions, with casts

```
/* strstack.c */
…
int StrStack_push(StrStack_T ss, const char *item) {
   return Stack_push(ss->s, (const void*)item);
}
char *StrStack_top(StrStack_T ss) {
   return (char*)Stack_top(ss->s);
}
void StrStack_pop(StrStack_T ss) {
   Stack_pop(ss->s);
}
int StrStack_isEmpty(StrStack_T ss) {
   return Stack_isEmpty(ss->s);
}
int StrStack_areEqual(StrStack_T ss1, StrStack_T ss2) {
   return Stack_areEqual(ss1->s, ss2->s,
      (int (*)(const void*, const void*))strcmp);
}
…
```

Appendix: The Wrapper Concept

- StrStack is a **wrapper** ADT
  - A StrStack object "wraps around" a Stack object
- A wrapper object
  - Does little work
  - Delegates (almost) all work to the wrapped object
- Pros and cons of the wrapper concept

  (+) **Type safety**: (As StrStack illustrates) wrapper can be designed to provide type safety

  (+) **Client convenience**: (More generally) wrapper tailors generic ADT to needs of specific client

  (-) **Developer inconvenience**: Must develop/maintain distinct wrapper for each distinct client need

Midterm Review
(overview of fall 2005 midterm)

Jennifer Rexford

1. Modulo Arithmetic and Character I/O

```
void f(unsigned int n) {
    do {
      putchar('0' + (n % 10));
    } while (n /= 10);
    putchar('\n');
}
```

- What does f(837) produce?
- What does this function do?

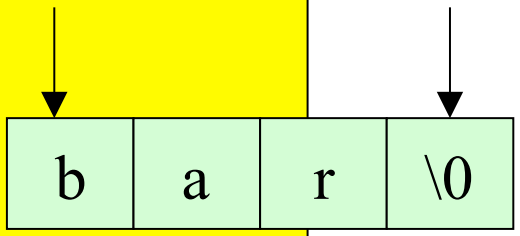
1. Modulo Arithmetic and Character I/O

```
void f(unsigned int n){
    for ( ; n; n /= 10)
        putchar('0' + (n % 10));
    putchar('\n');
}
```

- When is the answer different?

2. Pointers and Strings

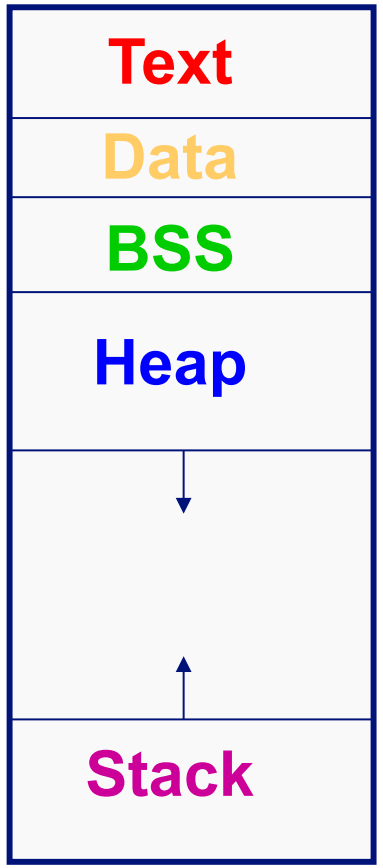
```
void f(char *s) {
  char *p = s;
  while (*s)
    s++;
  for (s--; s>p; s--,p++) {
    char c = *s;
    *s = *p;
    *p = c;
  }
}
```



- What does this function do?

3. Short Answer

- In the memory layout for a UNIX process:
  - Why does the heap grow from the top down and the stack from the bottom up, instead of both growing from the top down or both growing from the bottom up?



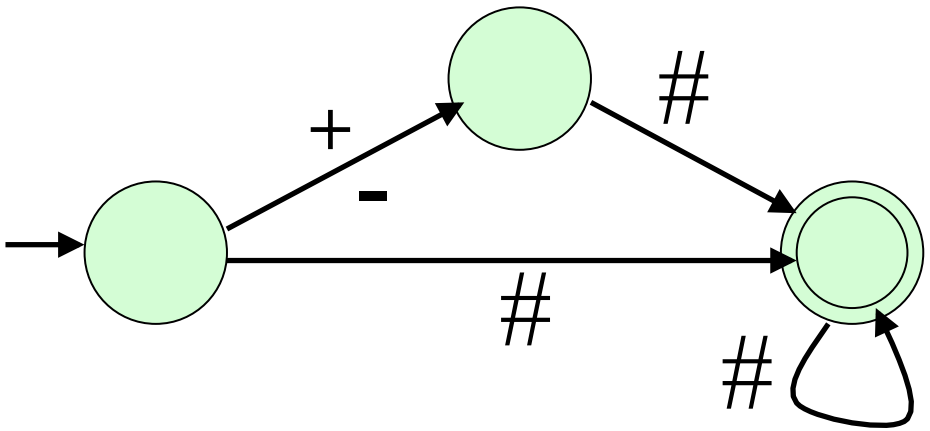
4. Deterministic Finite Automata

Identify whether or not a string is a floating-point number

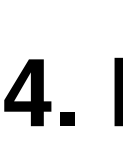
- Valid numbers
  - “-34”
  - “78.1”
  - “+298.3”
  - “-34.7e-1”
  - “34.7E-1”
  - “7.”
  - “.7”
  - “999.99e99”
- Invalid numbers
  - “abc”
  - “-e9”
  - “1e”
  - “+”
  - “17.9A”
  - “0.38+”
  - “.”
  - “38.38f9”

4. Deterministic Finite Automata

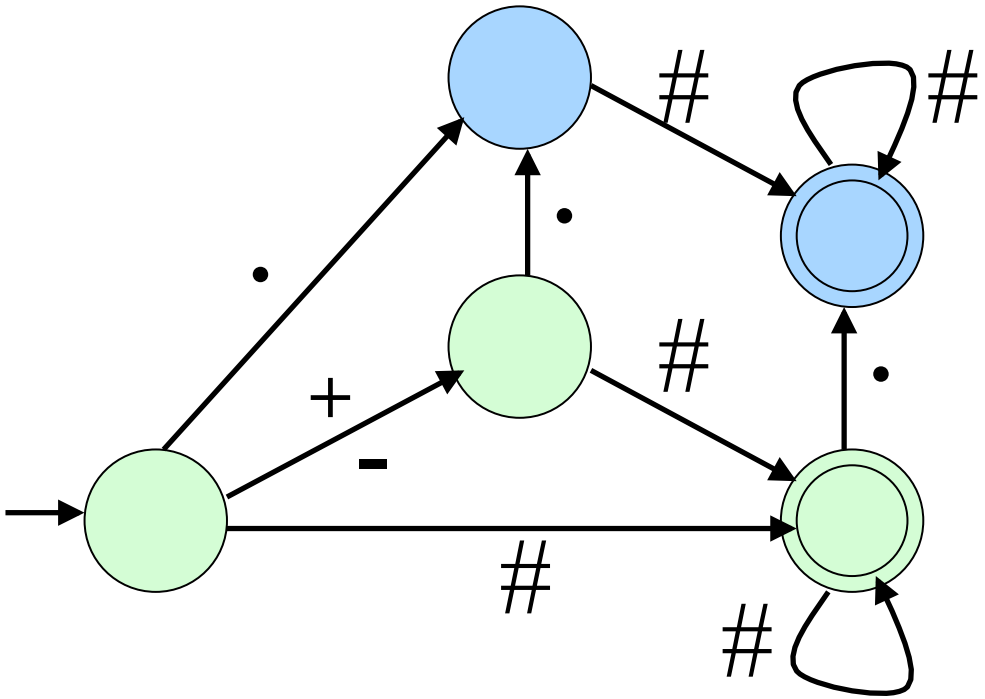
- Optional “+” or “-”
- Zero or more digits



4. Deterministic Finite Automata

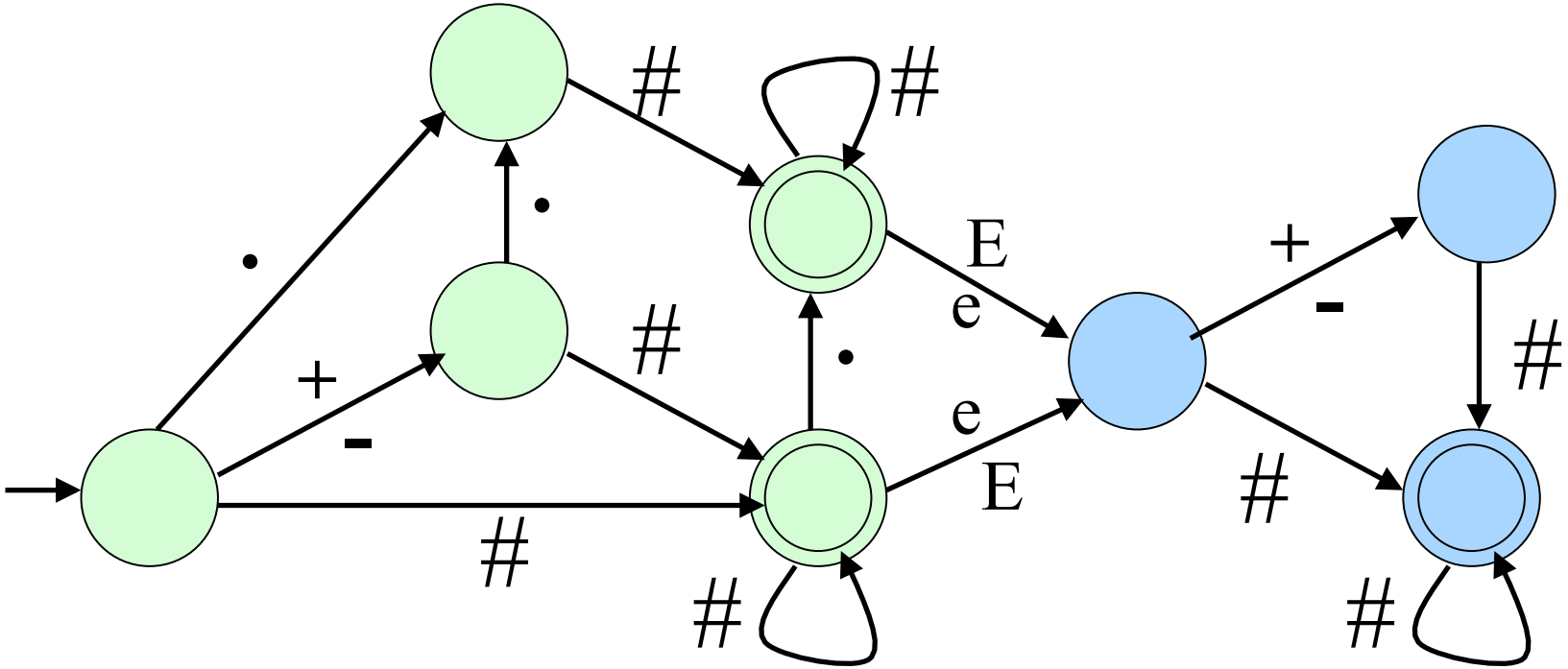


- Optional “+” or “-”
- Zero or more digits
- Optional decimal point
  - Followed by zero or more digits



4. Deterministic Finite Automata

- Optional "+" or "-"
- Zero or more digits
- Optional decimal point
  - Followed by zero or more digits
- Optional exponent "E" or "e"
  - Followed by optional "+" or "-"
  - Followed by one or more digits



5: Abstract Data Types

- Interface for a Queue (a first-in-first-out data structure)

```
#ifndef QUEUE_INCLUDED
#define QUEUE_INCLUDED
typedef struct Queue_t *Queue_T;


Queue_T Queue_new(void);
int Queue_empty(Queue_T queue);
void Queue_add(Queue_T queue, void* item);
void* Queue_remove(Queue_T queue);
#endif
```

5: Abstract Data Types

- An implementation for a Queue (in queue.c)

```
#include <stdlib.h>
#include <assert.h>
#include "queue.h"
struct list {
  void* item;
  struct list *next;
};
struct Queue_t {
  struct list *head;
  struct list *tail;
};
```

Why void*?

Why declared here and not in queue.h?

5: Abstract Data Types

- An implementation for a Queue_new

```
Queue_T Queue_new(void) {
  Queue_T queue = malloc(sizeof *queue);
  assert(queue != NULL);
  queue->head = NULL;
  queue->tail = NULL;
  return queue;
}
```

Implement a check for whether the queue is empty.

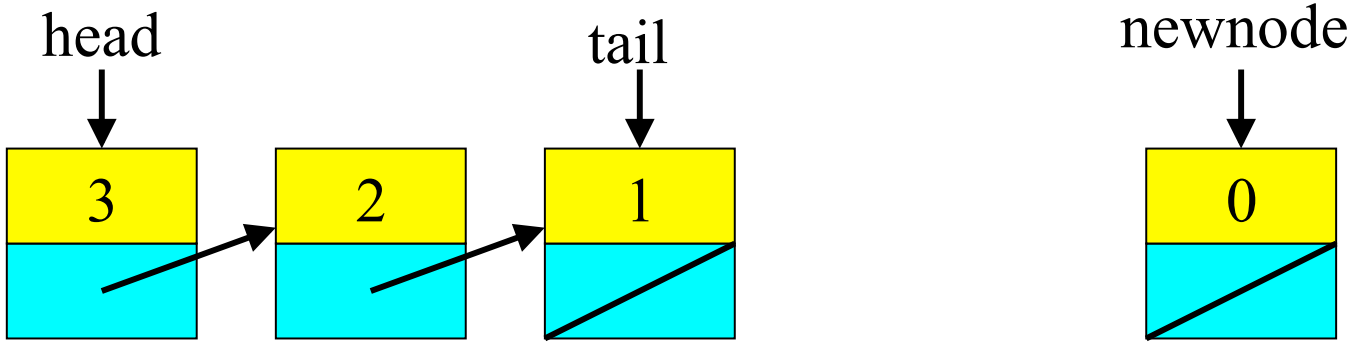
5: Abstract Data Types

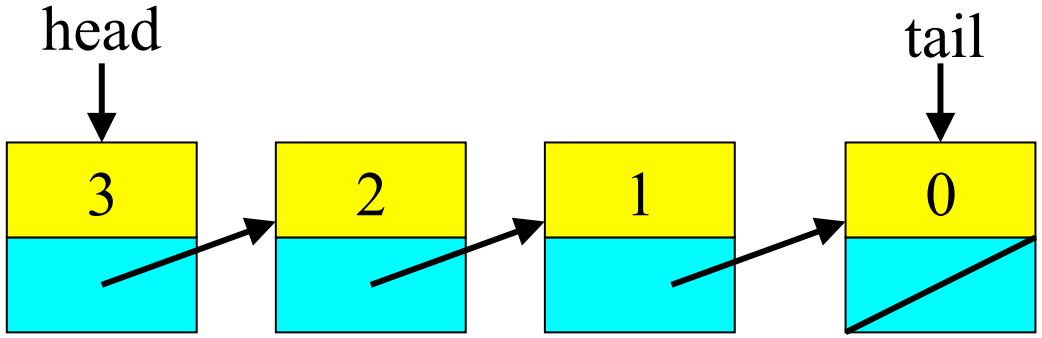
- An implementation for a **`Queue_empty`**

```
int Queue_empty(Queue_T queue) {
  assert(queue != NULL);
  return queue->head == NULL;
}
```

5: Abstract Data Types

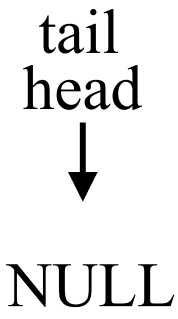
- An implementation for a **`Queue_add`**

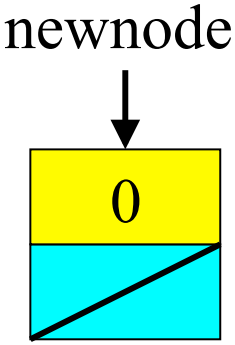


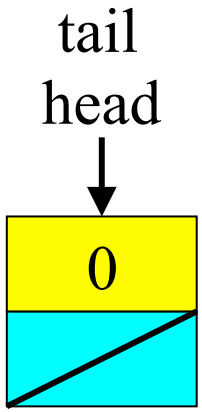


5: Abstract Data Types

- An implementation for a **`Queue_add`**







Queue_add() Implementation

```
void Queue_add(Queue_T queue, void *item) {
   struct list *newnode;
   assert(queue != NULL);
   newnode = (struct list*)malloc(sizeof(*newnode));
   assert(newnode != NULL);
   newnode->item = item;
   newnode->next = NULL;
   if (queue->tail == NULL)
      queue->head = newnode;
   else
      queue->tail->next = newnode;
   queue->tail = newnode;
}
```

5. ADT Common Mistakes

- Adding to the queue
  - Implementing a *stack* rather than a queue
    - Adding element to the head, rather than the tail
  - Not handling the case where the queue is empty
  - Missing assert() after call to malloc() for new entry
- Removing from the queue
  - Missing assert() when removing an element from an empty queue
  - Not handling removing the last item from the queue
  - Not doing a free() to return space used by the head element

Midterm Review
(overview of spring 2008 midterm)

Bit-Wise Manipulations

- Consider the following code, where **k** is an unsigned int:

printf("%u\n", k – ((k >> 2) << 2));

- What does the code do? Rewrite the line of code in a more efficient way.

- Replaces last two bits with 0
  - Same as doing: k & 3

What Does This Function Do?

```
char* f(unsigned int n) {
   int i, numbits = sizeof(unsigned int) * 8;
   char* ret = (char *) malloc(numbits + 1);
   for (i=numbits-1; i>=0; i--, n>>=1)
      ret[i] = '0' + (n & 1);
   ret[numbits] = '\0';
   return ret;
}
```

n = 19 00010011

Good Bug Hunting

- Consider this function that converts an integer to a string

```
char *itoa(int n) {
   char retbuf[5];
   sprintf(retbuf, "%d", n);
   return retbuf;
}
```

Not enough space

Temporary memory

- Where the **sprintf()** function "prints" to a formatted string, e.g., **sprintf(retbuf, "%d", 72)** places the string **"72"** starting at the location in memory indicated by the address **retbuf**:

Fixing the Bug: Rewrite

```
char *itoa(int n) {
   int size = 0;
   int temp = n;

   /* Count number of decimal digits in n */
   while (temp /= 10)
      size++;
   size++;

   /* If n is negative, add room for the "-" sign */
   if (n < 0)
      size++;
```

Fixing the Bug: Rewrite

```
  /* Allocate space for the string */
  char* retbuf = (char *) malloc(size + 1);
  assert(retbuf != NULL);

  /* Convert the number to a string of digits */
  sprintf(retbuf, "%d", n);

  return retbuf;
}
```

Preparing for the Exam

- Studying for the exam
  - Read through lecture and precept nodes
  - Study past midterm exams
  - Read through exercises in the book
- Taking the exam
  - Read briefly through all questions
  - Strategize where you spend your time
- Exam logistics
  - Wednesday 10-10:50am in COS 104
  - Open book, open notes, open mind… just no computer
  - No questions on UNIX tools (e.g., emacs, gcc, gdb, …)
- No Wednesday/Thursday precept this week
  - Have a great spring break!

Assembly Language: Overview

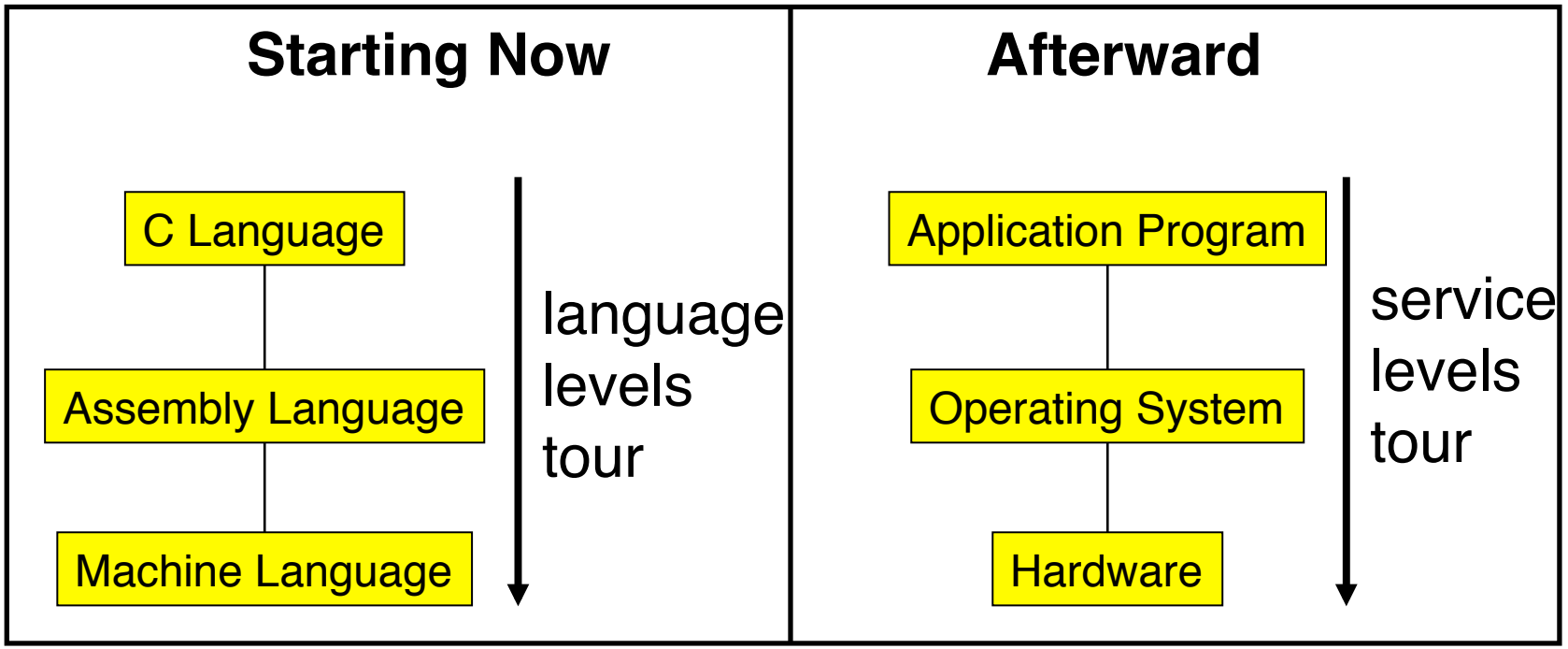
Jennifer Rexford

Goals of this Lecture

- Help you learn:
  - The basics of computer architecture
  - The relationship between C and assembly language
  - IA-32 assembly language, through an example

Context of this Lecture

Second half of the course



Three Levels of Languages

High-Level Language

- Make programming easier by describing operations in a semi-natural language
- Increase the portability of the code
- One line may involve many low-level operations
- Examples: C, C++, Java, Pascal, …

```
count = 0;
while (n > 1) {
   count++;
   if (n & 1)
      n = n*3 + 1;
   else
      n = n/2;
}
```

Assembly Language

- Tied to the specifics of the underlying machine
- Commands and names to make the code readable and writeable by humans
- Hand-coded assembly code may be more efficient
- E.g., IA-32 from Intel

```
        movl   $0, %ecx
loop:
        cmpl   $1, %edx
        jle    endloop
        addl   $1, %ecx
        movl   %edx, %eax
        andl   $1, %eax
        je     else
        movl   %edx, %eax
        addl   %eax, %edx
        addl   %eax, %edx
        addl   $1, %edx
        jmp    endif
else:
        sarl   $1, %edx
endif:
        jmp    loop
endloop:
```

Machine Language

- Also tied to the underlying machine
- What the computer sees and deals with
- Every instruction is a sequence of one or more numbers
- All stored in memory on the computer, and read and executed
- Unreadable by humans

```
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
9222 9120 1121 A120 1121 A121 7211 0000
0000 0001 0002 0003 0004 0005 0006 0007
0008 0009 000A 000B 000C 000D 000E 000F
0000 0000 0000 FE10 FACE CAFE ACED CEDE

1234 5678 9ABC DEF0 0000 0000 F00D 0000
0000 0000 EEEE 1111 EEEE 1111 0000 0000
B1B2 F1F5 0000 0000 0000 0000 0000 0000
```

Why Learn Assembly Language?

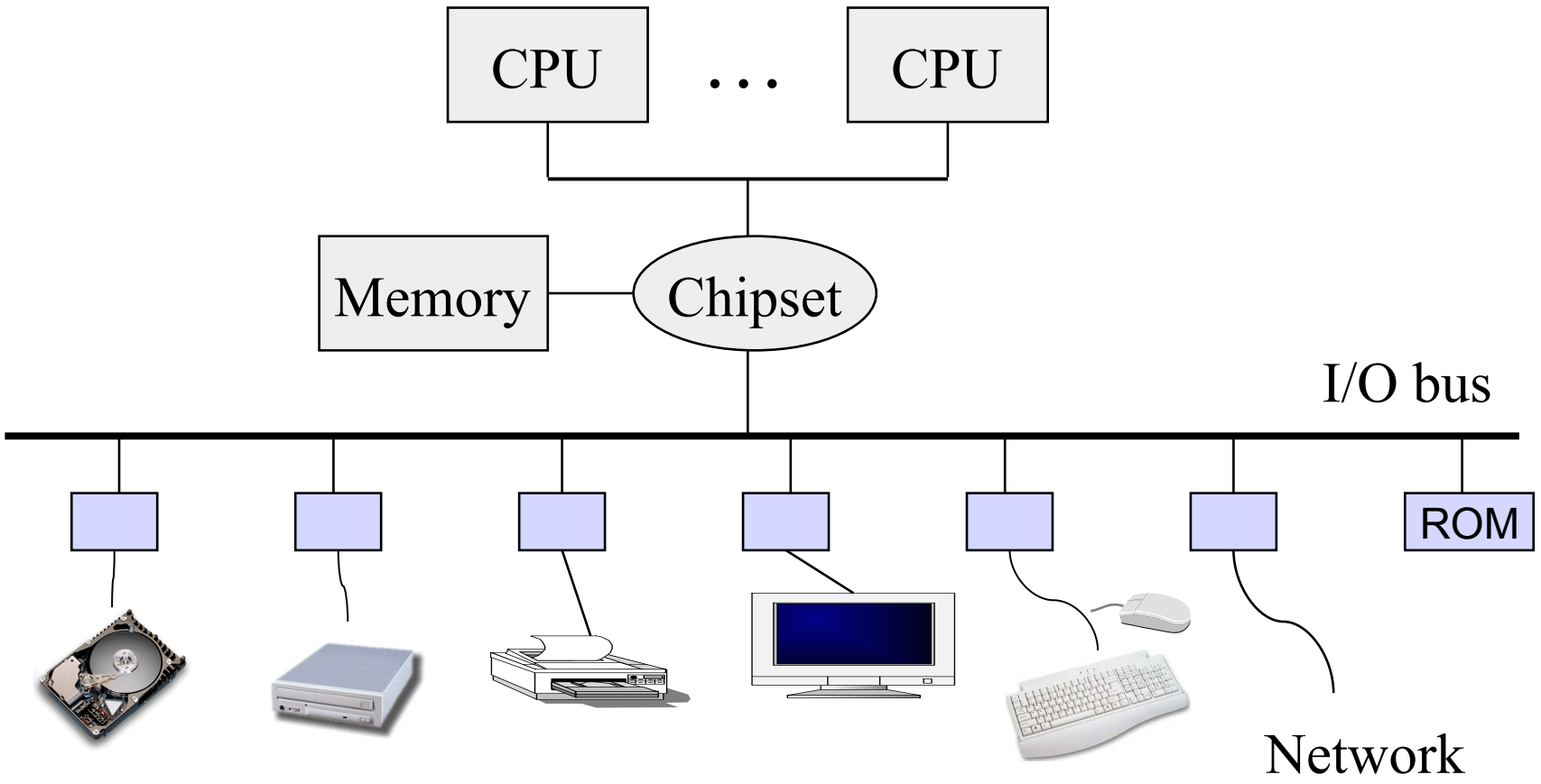
- Write faster code (even in high-level language)
  - By understanding which high-level constructs are better
  - … in terms of how efficient they are at the machine level
- Understand how things work underneath
  - Learn the basic organization of the underlying machine
  - Learn how the computer actually runs a program
  - Design better computers in the future
- Some software is still written in assembly language
  - Code that really needs to run quickly
  - Code for embedded systems, network processors, etc.

Why Learn Intel IA-32 Assembly?

- Program natively on our computing platform
  - Rather than using an emulator to mimic another machine
- Learn instruction set for the most popular platform
  - Most likely to work with Intel platforms in the future
- But, this comes at some cost in complexity
  - IA-32 has a large and varied set of instructions
  - More instructions than are really useful in practice
- Fortunately, you won't need to use everything

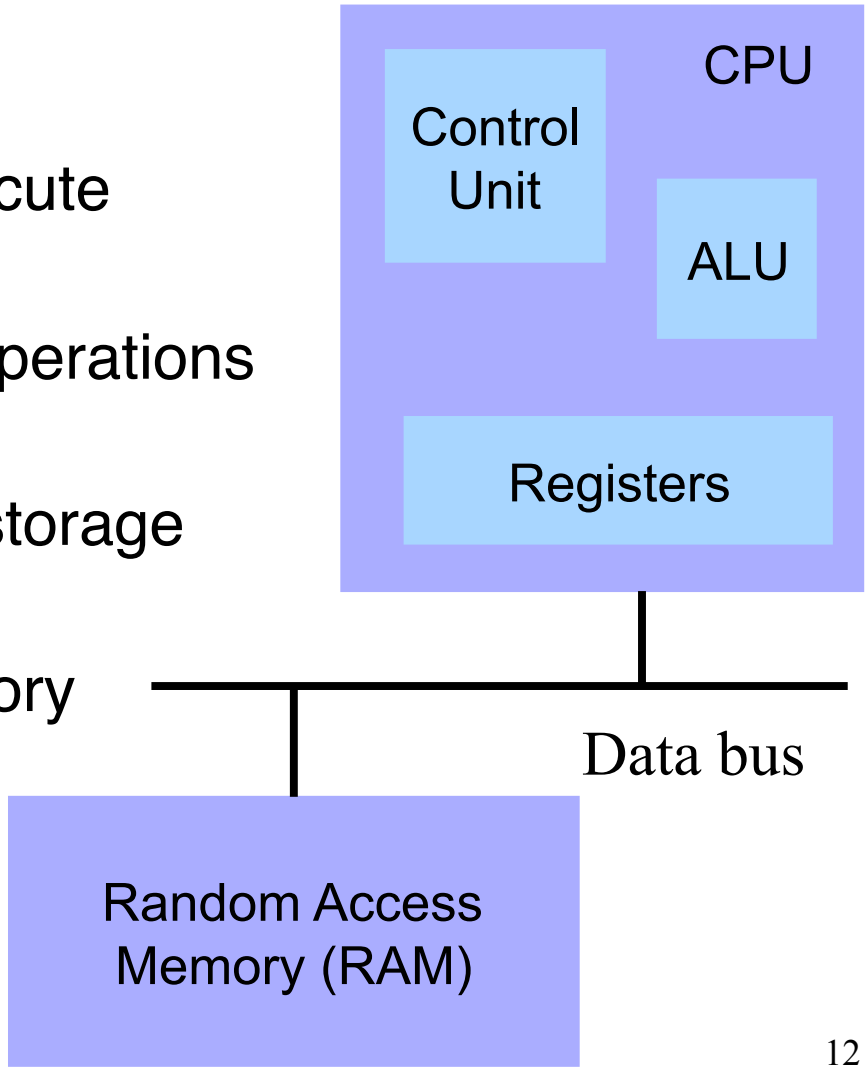
Computer Architecture

A Typical Computer



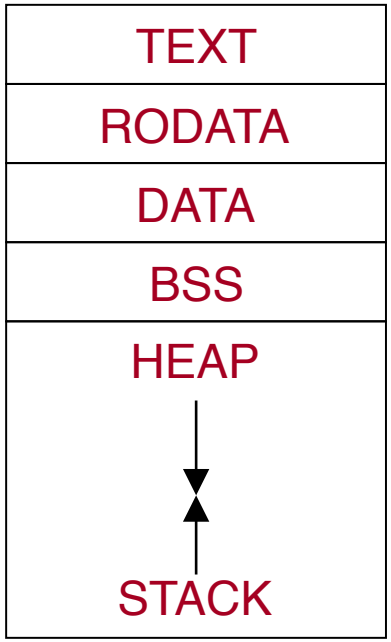
Von Neumann Architecture

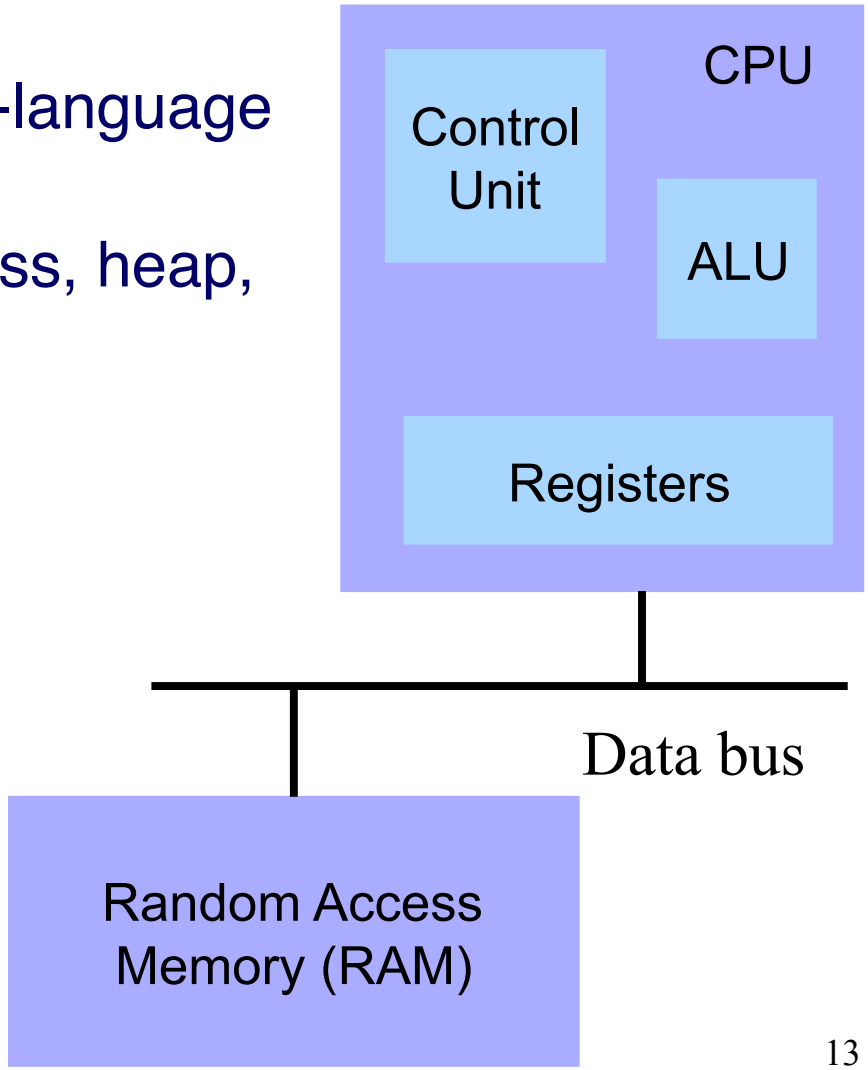
- Central Processing Unit
  - Control unit
    - Fetch, decode, and execute
  - Arithmetic and logic unit
    - Execution of low-level operations
  - General-purpose registers
    - High-speed temporary storage
  - Data bus
    - Provide access to memory



Von Neumann Architecture

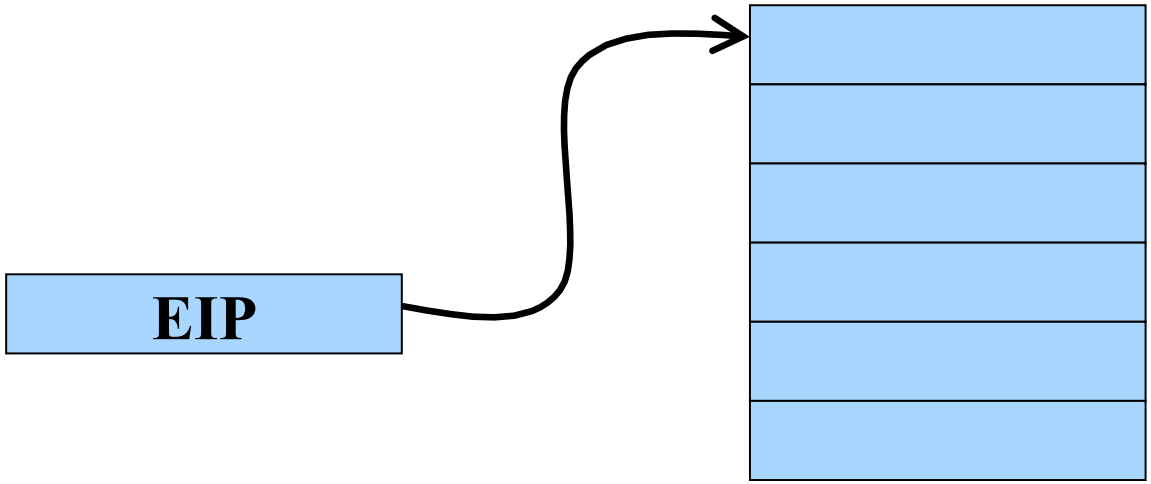
- Memory
  - Store executable machine-language instructions (text section)
  - Store data (rodata, data, bss, heap, and stack sections)





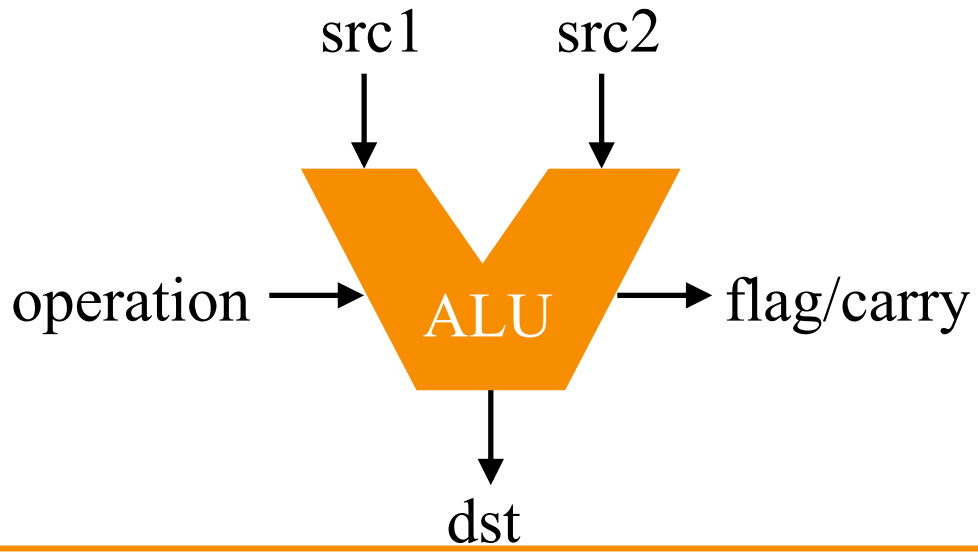
Control Unit: Instruction Pointer

- Stores the location of the next instruction
  - Address to use when reading machine-language instructions from memory (i.e., in the text section)
- Changing the instruction pointer (EIP)
  - Increment to go to the next instruction
  - Or, load a new value to "jump" to a new location



Control Unit: Instruction Decoder

- Determines what operations need to take place
  - Translate the machine-language instruction
- Control what operations are done on what data
  - E.g., control what data are fed to the ALU
  - E.g., enable the ALU to do multiplication or addition
  - E.g., read from a particular address in memory

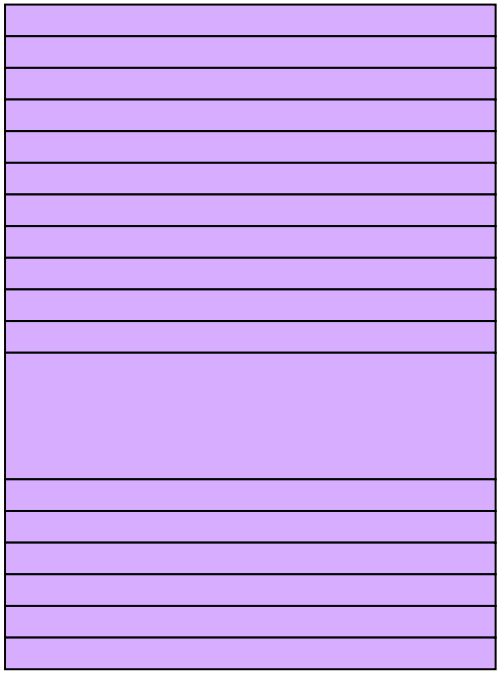


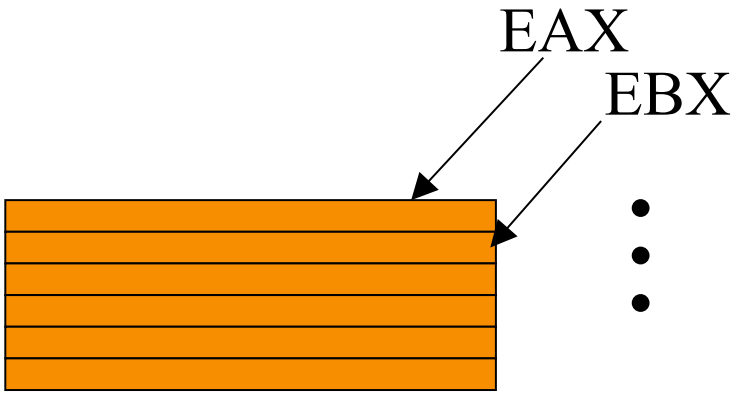
Registers

- Small amount of storage on the CPU
  - Can be accessed more quickly than main memory
- Instructions move data in and out of registers
  - Loading registers from main memory
  - Storing registers to main memory
- Instructions manipulate the register contents
  - Registers essentially act as temporary variables
  - For efficient manipulation of the data
- Registers are the top of the memory hierarchy
  - Ahead of main memory, disk, tape, …

Keeping it Simple: All 32-bit Words

- Simplifying assumption: all data in four-byte units
  - Memory is 32 bits wide
  - Registers are 32 bits wide





- In practice, can manipulate different sizes of data

C Code vs. Assembly Code

Kinds of Instructions

```
count = 0;
while (n > 1) {
   count++;
   if (n & 1)
      n = n*3 + 1;
   else
      n = n/2;
}
```

- **Reading and writing data**
  - count = 0
  - n
- **Arithmetic and logic operations**
  - Increment: count++
  - Multiply: n * 3
  - Divide: n/2
  - Logical AND: n & 1
- **Checking results of comparisons**
  - Is (n > 1) true or false?
  - Is (n & 1) non-zero or zero?
- **Changing the flow of control**
  - To the end of the while loop (if "n > 1")
  - Back to the beginning of the loop
  - To the else clause (if "n & 1" is 0)

Variables in Registers

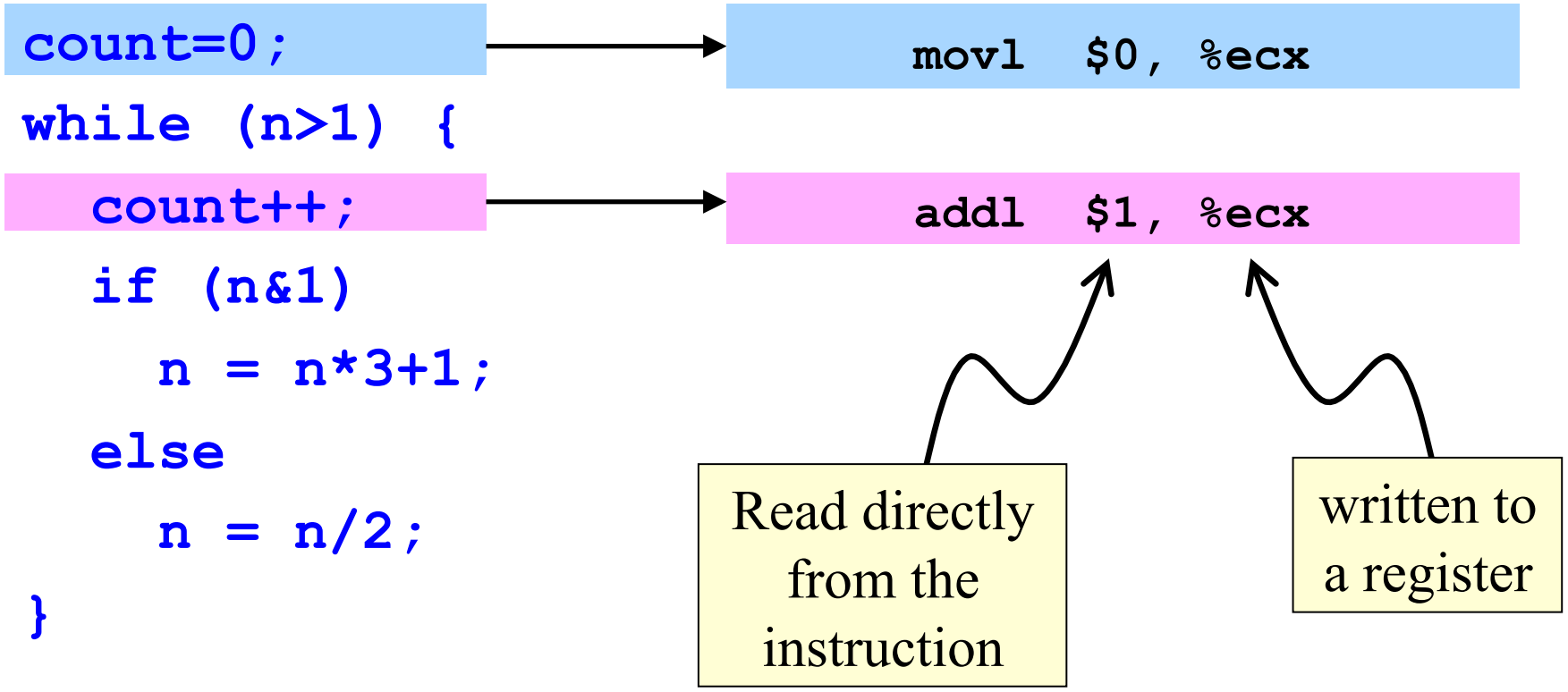
```
count = 0;
while (n > 1) {
   count++;
   if (n & 1)
      n = n*3 + 1;
   else
      n = n/2;
}
```

Registers

```
n     %edx
count %ecx
```

Referring to a register: percent sign ("%")

Immediate and Register Addressing



Referring to a immediate operand: dollar sign ("$")

Immediate and Register Addressing

```
count=0;
while (n>1) {
   count++;
   if (n&1)
      n = n*3+1;
   else
      n = n/2;
}
```

```
movl   %edx, %eax
andl   $1, %eax
```

Computing intermediate value in register EAX

Immediate and Register Addressing

```
count=0;
while (n>1) {
   count++;
   if (n&1)
      n = n*3+1;
   else
      n = n/2;
}
```

```
movl   %edx, %eax
addl   %eax, %edx
addl   %eax, %edx
addl   $1, %edx
```

Adding n twice is cheaper than multiplication!

Immediate and Register Addressing

```
count=0;
while (n>1) {
   count++;
   if (n&1)
      n = n*3+1;
   else
      n = n/2;
}
```

```
sarl   $1, %edx
```

Shifting right by 1 bit is cheaper than division!

Changing Program Flow

```
count=0;
while (n>1) {
   count++;
   if (n&1)
      n = n*3+1;
   else
      n = n/2;
}
```

- Cannot simply run next instruction
  - Check result of a previous operation
  - Jump to appropriate next instruction
- Flags register (EFLAGS)
  - Stores the status of operations, such as comparisons, as a side effect
  - E.g., last result was positive, negative, zero, etc.
- Jump instructions
  - Load new address in instruction pointer
- Example jump instructions
  - Jump unconditionally (e.g., "}")
  - Jump if zero (e.g., "n&1")
  - Jump if greater/less (e.g., "n>1")

Conditional and Unconditional Jumps

- Comparison **`cmpl`** compares two integers
  - Done by subtracting the first number from the second
    - Discarding the results, but setting flags as a side effect
  - Example:
    - **`cmpl $1, %edx`** (computes %edx – 1)
    - **`jle endloop`** (checks whether result was 0 or negative)
- Logical operation **`andl`** compares two integers
  - Example:
    - **`andl $1, %eax`** (bit-wise AND of %eax with 1)
    - **`je else`** (checks whether result was 0)
- Also, can do an unconditional branch **`jmp`**
  - Example:
    - **`jmp endif`** and **`jmp loop`**

Jump and Labels: While Loop

```
loop:
        cmpl   $1, %edx
        jle    endloop

        ...

while (n>1) {

}

        jmp    loop
endloop:
```

Checking if EDX is less than or equal to 1.

Jump and Labels: While Loop

```
count=0;
while (n>1) {
  count++;
  if (n&1)
    n = n*3+1;
  else
    n = n/2;
}
```

```
        movl   $0, %ecx
loop:
        cmpl   $1, %edx
        jle    endloop
        addl   $1, %ecx
        movl   %edx, %eax
        andl   $1, %eax
        je     else
        movl   %edx, %eax
        addl   %eax, %edx
        addl   %eax, %edx
        addl   $1, %edx
        jmp    endif
else:
        sarl   $1, %edx
endif:
        jmp    loop
endloop:
```

Jump and Labels: If-Then-Else

```
if (n&1)
   ...
else
   ...
```

```
        movl   %edx, %eax
        andl   $1, %eax
        je     else
        ...
        ...
        jmp    endif
else:
        ...
endif:
```

"then" block

"else" block

Jump and Labels: If-Then-Else

```
count=0;
while(n>1) {
  count++;
  if (n&1)
    n = n*3+1;
  else
    n = n/2;
}
```

```
        movl   $0, %ecx
loop:
        cmpl   $1, %edx
        jle    endloop
        addl   $1, %ecx
        movl   %edx, %eax
        andl   $1, %eax
        je     else
        movl   %edx, %eax
        addl   %eax, %edx
        addl   %eax, %edx
        addl   $1, %edx
        jmp    endif
else:
        sarl   $1, %edx
endif:
        jmp    loop
endloop:
```

"then" block

"else" block

Making the Code More Efficient…

```
count=0;
while(n>1) {
  count++;
  if (n&1)
    n = n*3+1;
  else
    n = n/2;
}
```

```
        movl   $0, %ecx
loop:
        cmpl   $1, %edx
        jle    endloop
        addl   $1, %ecx
        movl   %edx, %eax
        andl   $1, %eax
        je     else
        movl   %edx, %eax
        addl   %eax, %edx
        addl   %eax, %edx
        addl   $1, %edx
        jmp    endif
else:
        sarl   $1, %edx
endif:
        jmp    loop
endloop:
```

Replace with "jmp loop"

Complete Example

n %edx
count %ecx

```
count=0;
while (n>1) {
   count++;
   if (n&1)
      n = n*3+1;
   else
      n = n/2;
}
```

```
        movl   $0, %ecx
loop:
        cmpl   $1, %edx
        jle    endloop
        addl   $1, %ecx
        movl   %edx, %eax
        andl   $1, %eax
        je     else
        movl   %edx, %eax
        addl   %eax, %edx
        addl   %eax, %edx
        addl   $1, %edx
        jmp    endif
else:
        sarl   $1, %edx
endif:
        jmp    loop
endloop:
```

Reading IA-32 Assembly Language

- Referring to a register: percent sign ("%")
  - E.g., "%ecx" or "%eip"
- Referring to immediate operand: dollar sign ("$")
  - E.g., "$1" for the number 1
- Storing result: typically in the second argument
  - E.g. "addl $1, %ecx" increments register ECX
  - E.g., "movl %edx, %eax" moves EDX to EAX
- Assembler directives: starting with a period (".")
  - E.g., ".section .text" to start the text section of memory
- Comment: pound sign ("#")
  - E.g., "# Purpose: Convert lower to upper case"

Conclusions

- Assembly language
  - In between high-level language and machine code
  - Programming the “bare metal” of the hardware
  - Loading and storing data, arithmetic and logic operations, checking results, and changing control flow
- To get more familiar with IA-32 assembly
  - Read more assembly-language examples
    - Chapter 3 of Bryant and O’Hallaron book
  - Generate your own assembly-language code
    - gcc217 –S –O2 code.c

Assembly Language: IA-32 Instructions

Jennifer Rexford

Goals of this Lecture

- Help you learn how to:
  - Manipulate data of various sizes
  - Leverage more sophisticated addressing modes
  - Use condition codes and jumps to change control flow
- So you can:
  - Write more efficient assembly-language programs
  - Understand the relationship to data types and common programming constructs in high-level languages
- Focus is on the assembly-language code
  - Rather than the layout of memory for storing data

Variable Sizes in High-Level Language

- C data types vary in size
  - Character: 1 byte
  - Short, int, and long: varies, depending on the computer
  - Float and double: varies, depending on the computer
  - Pointers: typically 4 bytes
- Programmer-created types
  - Struct: arbitrary size, depending on the fields
- Arrays
  - Multiple consecutive elements of some fixed size
  - Where each element could be a struct

Supporting Different Sizes in IA-32

- Three main data sizes
  - Byte (b): 1 byte
  - Word (w): 2 bytes
  - Long (l): 4 bytes
- Separate assembly-language instructions
  - E.g., addb, addw, and addl
- Separate ways to access (parts of) a register
  - E.g., %ah or %al, %ax, and %eax
- Larger sizes (e.g., struct)
  - Manipulated in smaller byte, word, or long units

Byte Order in Multi-Byte Entities

- Intel is a little endian architecture
  - Least significant byte of multi-byte entity is stored at lowest memory address
  - "Little end goes first"

The int 5 at address 1000:

1000	00000101
1001	00000000
1002	00000000
1003	00000000

- Some other systems use big endian
  - Most significant byte of multi-byte entity is stored at lowest memory address
  - "Big end goes first"

The int 5 at address 1000:

1000	00000000
1001	00000000
1002	00000000
1003	00000101

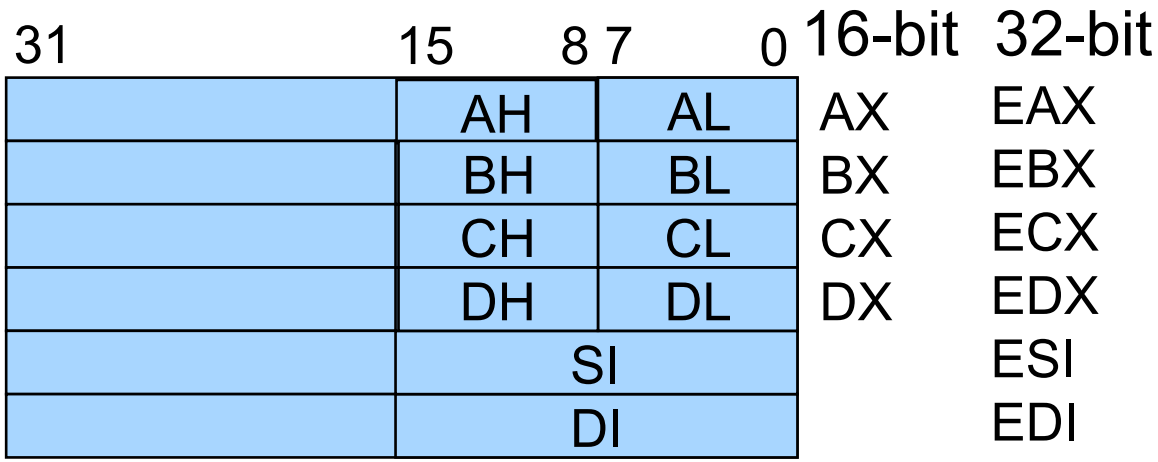
Little Endian Example

```
int main(void) {
  int i=0x003377ff, j;
  unsigned char *p = (unsigned char *) &i;

  for (j=0; j<4; j++)
    printf("Byte %d: %x\n", j, p[j]);
}
```

Output on a little-endian machine

Byte 0: ff
Byte 1: 77
Byte 2: 33
Byte 3: 0

IA-32 General Purpose Registers

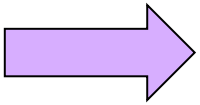


General-purpose registers

C Example: One-Byte Data

**Global *char* variable i is in *%al*,
the *lower byte* of the "A" register.**

```
char i;
…
if (i > 5) {
   i++;
else
   i--;
}
```

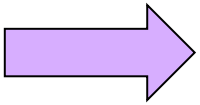


```
      cmpb $5, %al
      jle else
      incb %al
      jmp endif
else:
      decb %al
endif:
```

C Example: Four-Byte Data

Global *int* variable i is in *%eax*,
the *full 32 bits* of the "A" register.

```
int i;
…
if (i > 5) {
   i++;
else
   i--;
}
```



```
      cmpl $5, %eax
      jle else
      incl %eax
      jmp endif
else:
      decl %eax
endif:
```

Loading and Storing Data

- Processors have many ways to access data
  - Known as “addressing modes”
  - Two simple ways seen in previous examples
- Immediate addressing
  - Example: movl **$0**, %ecx
  - Data (e.g., number “0”) embedded in the instruction
  - Initialize register ECX with zero
- Register addressing
  - Example: movl **%edx**, **%ecx**
  - Choice of register(s) embedded in the instruction
  - Copy value in register EDX into register ECX

Accessing Memory

- Variables are stored in memory
  - Global and static local variables in Data or BSS section
  - Dynamically allocated variables in the heap
  - Function parameters and local variables on the stack
- Need to be able to load from and store to memory
  - To manipulate the data directly in memory
  - Or copy the data between main memory and registers
- IA-32 has many different addressing modes
  - Corresponding to common programming constructs
  - E.g., accessing a global variable, dereferencing a pointer, accessing a field in a struct, or indexing an array

Direct Addressing

- Load or store from a particular memory location
  - Memory address is embedded in the instruction
  - Instruction reads from or writes to that address
- IA-32 example: movl **2000**, %ecx
  - Four-byte variable located at address 2000
  - Read four bytes starting at address 2000
  - Load the value into the ECX register
- Useful when the address is known in advance
  - Global variables in the Data or BSS sections
- Can use a label for (human) readability
  - E.g., “i” to allow “movl i, %eax”

Indirect Addressing

- Load or store from a previously-computed address
  - Register with the address is embedded in the instruction
  - Instruction reads from or writes to that address
- IA-32 example: movl **(%eax)**, %ecx
  - EAX register stores a 32-bit address (e.g., 2000)
  - Read long-word variable stored at that address
  - Load the value into the ECX register
- Useful when address is not known in advance
  - Dynamically allocated data referenced by a pointer
  - The “(%eax)” essentially dereferences a pointer

Base Pointer Addressing

- Load or store with an offset from a base address
  - Register storing the base address
  - Fixed offset also embedded in the instruction
  - Instruction computes the address and does access
- IA-32 example: movl 8(%eax), %ecx
  - EAX register stores a 32-bit base address (e.g., 2000)
  - Offset of 8 is added to compute address (e.g., 2008)
  - Read long-word variable stored at that address
  - Load the value into the ECX register
- Useful when accessing part of a larger variable
  - Specific field within a “struct”
  - E.g., if “age” starts at the 8th byte of “student” record

Indexed Addressing

- Load or store with an offset and multiplier
  - Fixed based address embedded in the instruction
  - Offset computed by multiplying register with constant
  - Instruction computes the address and does access
- IA-32 example: movl 2000(,%eax,4), %ecx
  - Index register EAX (say, with value of 10)
  - Multiplied by a multiplier of 1, 2, 4, or 8 (say, 4)
  - Added to a fixed base of 2000 (say, to get 2040)
- Useful to iterate through an array (e.g., a[i])
  - Base is the start of the array (i.e., “a”)
  - Register is the index (i.e., “i”)
  - Multiplier is the size of the element (e.g., 4 for “int”)

Indexed Addressing Example

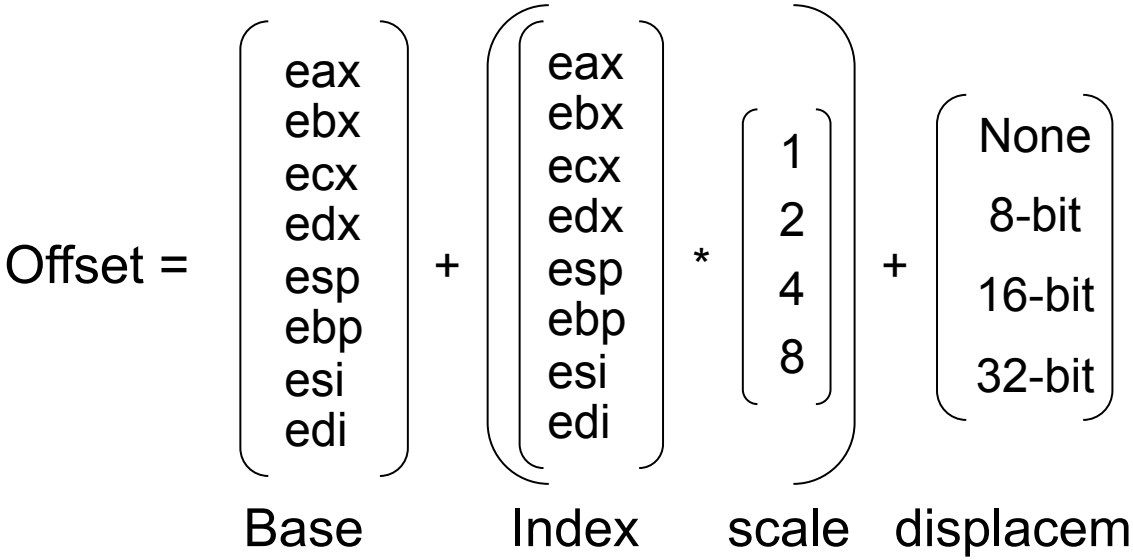
```
int a[20];
...
int i, sum=0;
for (i=0; i<20; i++)
    sum += a[i];
```

global variable

EAX: i
EBX: sum
ECX: temporary

```
    movl $0, %eax
    movl $0, %ebx
sumloop:
    movl a(,%eax,4), %ecx
    addl %ecx, %ebx
    incl %eax
    cmpl $19, %eax
    jle sumloop
```

Effective Address: More Generally



- Displacement
- Base
- Base + displacement
- (Index * scale) + displacement
- Base + (index * scale) + displacement

```
movl foo, %ebx
movl (%eax), %ebx
movl foo(%eax), %ebx
movl 1(%eax), %ebx
movl (,%eax,4), %ebx
movl foo(%edx,%eax,4),%ebx
```

Data Access Methods: Summary

- Immediate addressing: data stored in the instruction itself
  - **movl $10, %ecx**
- Register addressing: data stored in a register
  - **movl %eax, %ecx**
- Direct addressing: address stored in instruction
  - **movl foo, %ecx**
- Indirect addressing: address stored in a register
  - **movl (%eax), %ecx**
- Base pointer addressing: includes an offset as well
  - **movl 4(%eax), %ecx**
- Indexed addressing: instruction contains base address, and specifies an index register and a multiplier (1, 2, 4, or 8)
  - **movl 2000(,%eax,1), %ecx**

Control Flow

- Common case
  - Execute code sequentially
  - One instruction after another
- Sometimes need to change control flow
  - If-then-else
  - Loops
  - Switch
- Two key ingredients
  - Testing a condition
  - Selecting what to run next based on result

```
        cmpl $5, %eax
        jle else
        incl %eax
        jmp endif
else:
        decl %eax
endif:
```

Condition Codes

- 1-bit registers set by arithmetic & logic instructions
  - ZF: Zero Flag
  - SF: Sign Flag
  - CF: Carry Flag
  - OF: Overflow Flag
- Example: "addl Src, Dest" ("t = a + b")
  - ZF: set if t == 0
  - SF: set if t < 0
  - CF: set if carry out from most significant bit
    - *Unsigned* overflow
  - OF: set if two's complement overflow
    - (a>0 && b>0 && t<0)
      ll (a<0 && b<0 && t>=0)

Condition Codes (continued)

- Example: "cmpl Src2,Src1" (compare b,a)
  - Like computing a-b without setting destination
  - ZF: set if a == b
  - SF: set if (a-b) < 0
  - CF: set if carry out from most significant bit
    - Used for unsigned comparisons
  - OF: set if two's complement overflow
    - (a>0 && b<0 && (a-b)<0) ll (a<0 && b>0 && (a-b)>0)
- Flags are *not* set by lea, inc, or dec instructions
  - Hint: this is useful in the assembly-language programming assignment! ☺

Example Five-Bit Comparisons

- Comparison: cmp $6, $12
  - Not zero: ZF=0 (diff is not 00000)
  - Positive: SF=0 (first bit is 0)
  - No carry: CF=0 (unsigned diff is correct)
  - No overflow: OF=0 (signed diff is correct)

$$\begin{array}{r} 01100 \\ -\ 00110 \\ \hline ?? \end{array} \longrightarrow \begin{array}{r} 01100 \\ +11010 \\ \hline 00110 \end{array}$$

- Comparison: cmp $12, $6
  - Not zero: ZF=0 (diff is not 00000)
  - Negative: SF=1 (first bit is 1)
  - Carry: CF=1 (unsigned diff is wrong)
  - No overflow: OF=0 (signed diff is correct)

$$\begin{array}{r} 00110 \\ -\ 01100 \\ \hline ?? \end{array} \longrightarrow \begin{array}{r} 00110 \\ +10100 \\ \hline 11010 \end{array}$$

- Comparison: cmp $-6, $-12
  - Not zero: ZF=0 (diff is not 00000)
  - Negative: SF=1 (first bit is 1)
  - Carry: CF=1 (unsigned diff of 20 and 28 is wrong)
  - No overflow: OF=0 (signed diff is correct)

$$\begin{array}{r} 10100 \\ -\ 11010 \\ \hline ?? \end{array} \longrightarrow \begin{array}{r} 10100 \\ +00110 \\ \hline 11010 \end{array}$$

Jumps after Comparison (cmpl)

- Equality
  - Equal: je (ZF)
  - Not equal: jne (~ZF)
- Below/above (e.g., unsigned arithmetic)
  - Below: jb (CF)
  - Above or equal: jae (~CF)
  - Below or equal: jbe (CF | ZF)
  - Above: ja (~(CF | ZF))
- Less/greater (e.g., signed arithmetic)
  - Less: jl (SF ^ OF)
  - Greater or equal: jge (~(SF ^ OF))
  - Less or equal: jle ((SF ^ OF) | ZF)
  - Greater: jg (!((SF ^ OF) | ZF))

Branch Instructions

- Conditional jump
  - j{l,g,e,ne,...} target if (condition) {eip = target}

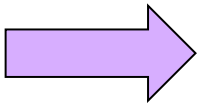
Comparison	Signed	Unsigned	
=	e	e	*"equal"*
≠	ne	ne	*"not equal"*
>	g	a	*"greater,above"*
≥	ge	ae	*"...-or-equal"*
<	l	b	*"less,below"*
≤	le	be	*"...-or-equal"*
overflow/carry	o	c	
no ovf/carry	no	nc	

- Unconditional jump
  - jmp target
  - jmp *register

Jumping

- Simple model of a "goto" statement
  - Go to a particular place in the code
  - Based on whether a condition is true or false
  - Can represent if-the-else, switch, loops, etc.
- Pseudocode example: If-Then-Else

```
if (Test) {
    then-body;
} else {
    else-body;
```

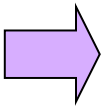


```
  if (!Test) jump to Else;
  then-body;
  jump to Done;
Else:
  else-body;
Done:
```

Jumping (continued)

- Pseudocode example: Do-While loop

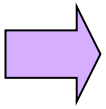
```
do {
  Body;
} while (Test);
```



```
loop:
  Body;
  if (Test) then jump to loop;
```

- Pseudocode example: While loop

```
while (Test)
  Body;
```

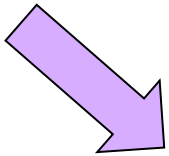


```
   jump to middle;
loop:
   Body;
middle:
   if (Test) then jump to loop;
```

Jumping (continued)

- Pseudocode example: For loop

```
for (Init; Test; Update)
    Body
```



```
    Init;
    if (!Test) jump to done;
loop:
    Body;
    Update;
    if (Test) jump to loop;
done:
```

Arithmetic Instructions

- Simple instructions
  - add{b,w,l} source, dest — dest = source + dest
  - sub{b,w,l} source, dest — dest = dest – source
  - Inc{b,w,l} dest — dest = dest + 1
  - dec{b,w,l} dest — dest = dest – 1
  - neg{b,w,l} dest — dest = ~dest + 1
  - cmp{b,w,l} source1, source2 — source2 – source1
- Multiply
  - mul (unsigned) or imul (signed)

```
mull %ebx          # edx, eax = eax * ebx
```

- Divide
  - div (unsigned) or idiv (signed)

```
idiv %ebx          # edx = edx,eax / ebx
```

- Many more in Intel manual (volume 2)
  - adc, sbb, decimal arithmetic instructions

Bitwise Logic Instructions

- Simple instructions

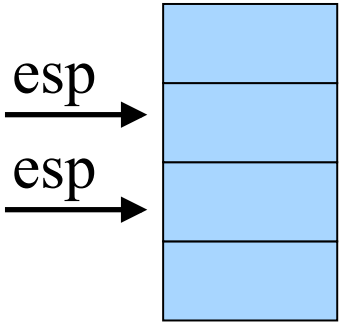
and{b,w,l} source, dest	dest = source & dest
or{b,w,l} source, dest	dest = source | dest
xor{b,w,l} source, dest	dest = source ^ dest
not{b,w,l} dest	dest = ~dest
sal{b,w,l} source, dest (arithmetic)	dest = dest << source
sar{b,w,l} source, dest (arithmetic)	dest = dest >> source

- Many more in Intel Manual (volume 2)
  - Logic shift
  - Rotation shift
  - Bit scan
  - Bit test
  - Byte set on conditions

Data Transfer Instructions

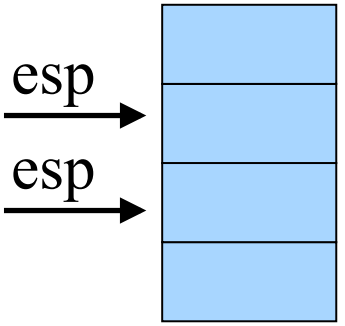
- **`mov{b,w,l} source, dest`**
  - General move instruction
- **`push{w,l} source`**

```
pushl %ebx    # equivalent instructions
              subl $4, %esp
              movl %ebx, (%esp)
```



- **`pop{w,l} dest`**

```
popl %ebx     # equivalent instructions
              movl (%esp), %ebx
              addl  $4, %esp
```



- Many more in Intel manual (volume 2)
  - Type conversion, conditional move, exchange, compare and exchange, I/O port, string move, etc.

Conclusions

- Accessing data
  - Byte, word, and long-word data types
  - Wide variety of addressing modes
- Control flow
  - Common C control-flow constructs
  - Condition codes and jump instructions
- Manipulating data
  - Arithmetic and logic operations
- Next time
  - Calling functions, using the stack

Assembly Language: Function Calls

Jennifer Rexford

Goals of this Lecture

- Function call problems:
  - Calling and returning
  - Passing parameters
  - Storing local variables
  - Handling registers without interference
  - Returning values
- IA-32 solutions to those problems
  - Pertinent instructions and conventions

Recall from Last Lecture

Examples of Operands

- Immediate Operand
  - movl $5, …
    - CPU uses 5 as source operand
  - movl $i, …
    - CPU uses address denoted by i as source operand
- Register Operand
  - movl %eax, …
    - CPU uses contents of EAX register as source operand

Recall from Last Lecture (cont.)

- Memory Operand: Direct Addressing
  - movl i, …
    - CPU fetches source operand from memory at address i
- Memory Operand: Indirect Addressing
  - movl (%eax), …
    - CPU considers contents of EAX to be an address
    - Fetches source operand from memory at that address
- Memory Operand: Base+Displacement Addressing
  - movl 8(%eax), …
    - CPU computes address as 8 + [contents of EAX]
    - Fetches source operand from memory at that address

Recall from Last Lecture (cont.)

- Memory Operand: Indexed Addressing
  - movl 8(%eax, %ecx), …
    - Computes address as 8 + [contents of EAX] + [contents of ECX]
    - Fetches source operand from memory at that address
- Memory Operand: Scaled Indexed Addressing
  - movl 8(%eax, %ecx, 4), …
    - Computes address as 8 + [contents of EAX] + ([contents of ECX] * 4)
    - Fetches source operand from memory at that address
- Same for destination operand, except…
- Destination operand cannot be immediate

Function Call Problems

1. Calling and returning
   - How does caller function *jump* to callee function?
   - How does callee function *jump back* to the right place in caller function?
2. Passing parameters
   - How does caller function pass *parameters* to callee function?
3. Storing local variables
   - Where does callee function store its *local variables?*
4. Handling registers
   - How do caller and callee functions use *same registers* without interference?
5. Returning a value
   - How does callee function send *return value* back to caller function?

Problem 1: Calling and Returning

How does caller function *jump* to callee function?

- I.e., Jump to the address of the callee's first instruction

How does the callee function *jump back* to the right place in caller function?

- I.e., Jump to the instruction immediately following the most-recently-executed call instruction

Attempted Solution: Use Jmp Instruction

- Attempted solution: caller and callee use jmp instruction

```
P:                  # Function P
   …
   jmp R            # Call R
Rtn_point1:
   …
```

```
R:                  # Function R
   …
   jmp Rtn_point1   # Return
```

Attempted Solution: Use Jmp Instruction

- Problem: callee may be called by multiple callers

```
P:              # Function P
   …
   jmp R        # Call R
Rtn_point1:
   …
```

```
R:              # Function R
   …
   jmp ???      # Return
```

```
Q:              # Function Q
   …
   jmp R        # Call R
Rtn_point2:
   …
```

Attempted Solution: Use Register

- Attempted solution 2: Store return address in register

```
P:              # Function P
   movl $Rtn_point1, %eax
   jmp R        # Call R
Rtn_point1:
   …
```

```
Q:              # Function Q
   movl $Rtn_point2, %eax
   jmp R        # Call R
Rtn_point2:
   …
```

```
R:              # Function R
   …
   jmp *%eax    # Return
```

Special form of jmp instruction; we will not use

Attempted Solution: Use Register

- Problem: Cannot handle nested function calls

```
P:              # Function P
   movl $Rtn_point1, %eax
   jmp Q        # Call Q
Rtn_point1:
   …
```

```
R:              # Function R
   …
   jmp *%eax    # Return
```

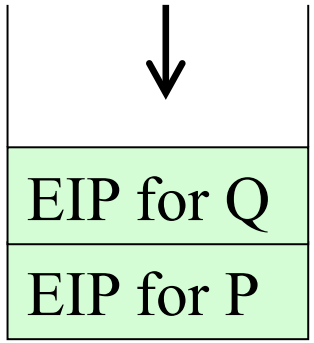
```
Q:              # Function Q
   movl $Rtn_point2, %eax
   jmp R        # Call R
Rtn_point2:
   …
   jmp %eax     # Return
```

Problem if P calls Q, and Q calls R

Return address for P to Q call is lost

IA-32 Solution: Use the Stack

- May need to store many return addresses
  - The number of nested functions is not known in advance
  - A return address must be saved for as long as the function invocation continues, and discarded thereafter
- Addresses used in reverse order
  - E.g., function P calls Q, which then calls R
  - Then R returns to Q which then returns to P



- Last-in-first-out data structure (stack)
  - Caller pushes return address on the stack
  - … and callee pops return address off the stack
- IA 32 solution: Use the stack via call and ret

IA-32 Call and Ret Instructions

- Ret instruction "knows" the return address

```
P:                  # Function P

    …
    call R
    call Q
    …
```

```
R:                  # Function R

    …
    ret
```

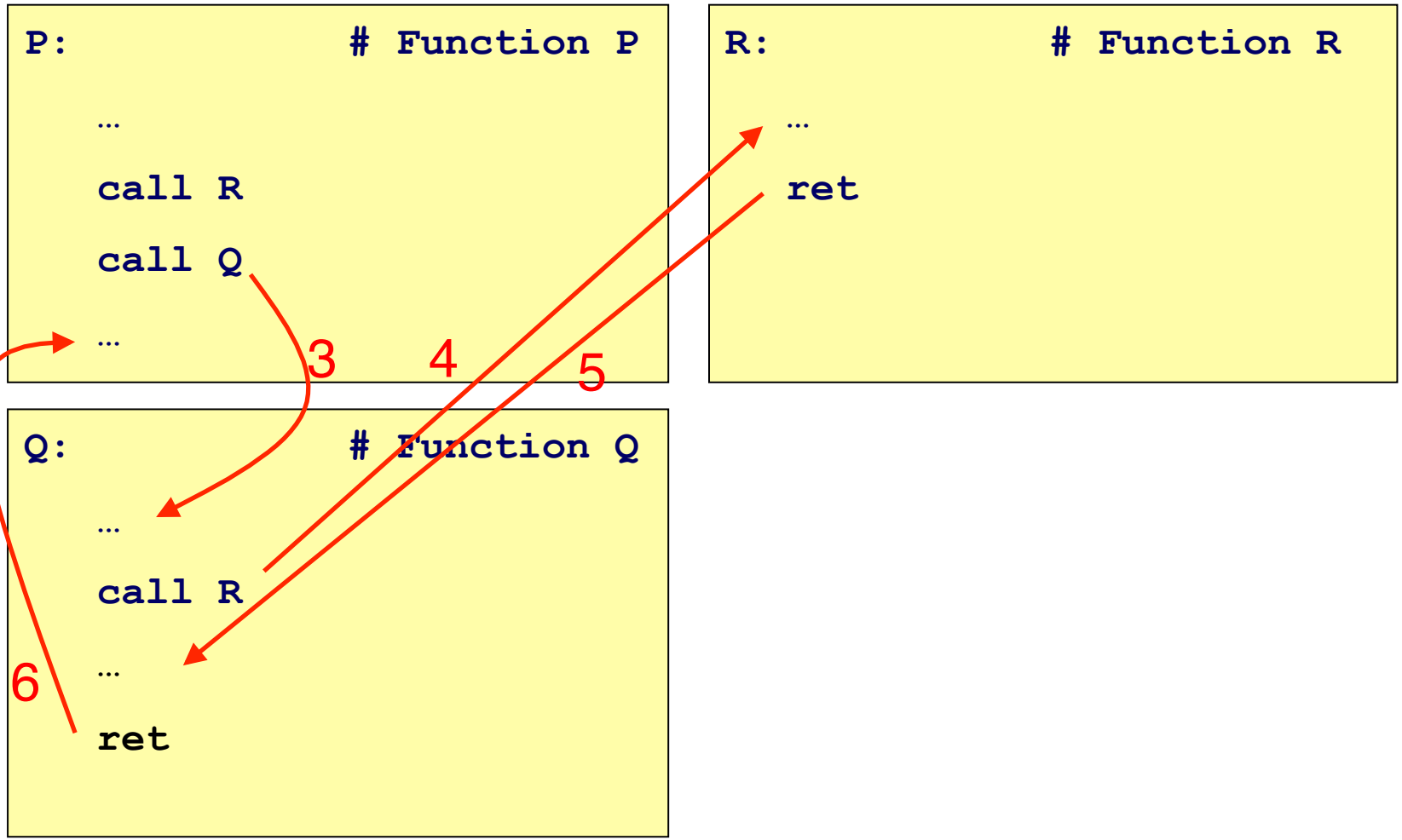
1

2

```
Q:                  # Function Q

    …
    call R
    …
    ret
```

IA-32 Call and Ret Instructions

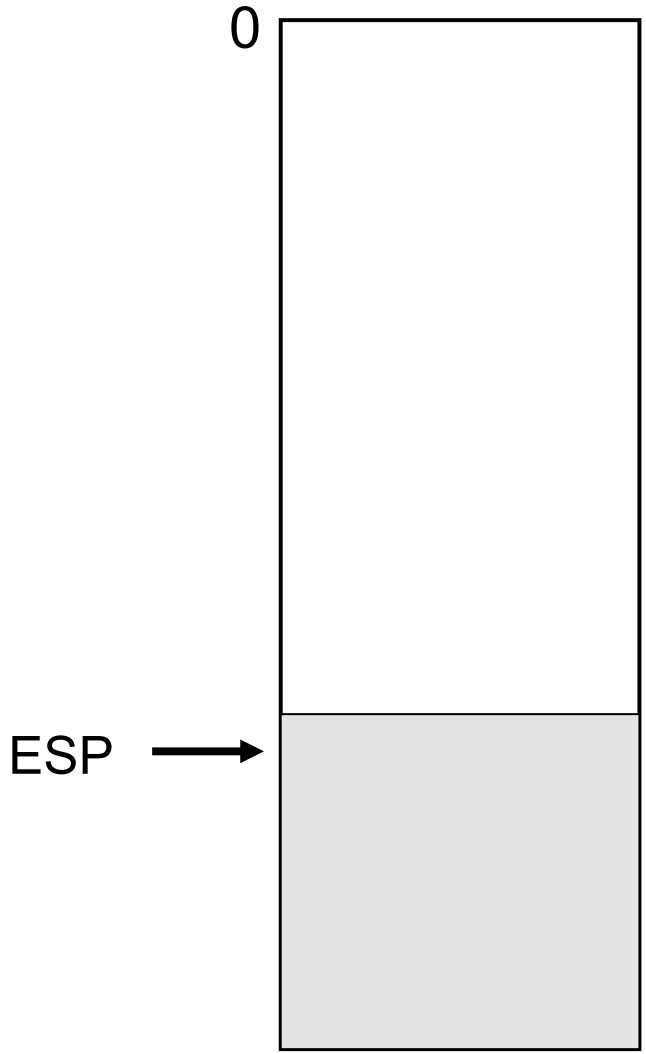
- Ret instruction "knows" the return address



Implementation of Call

- ESP (stack pointer register) points to top of stack

Instruction	Effective Operations
`pushl src`	`subl $4, %esp` `movl src, (%esp)`
`popl dest`	`movl (%esp), dest` `addl $4, %esp`

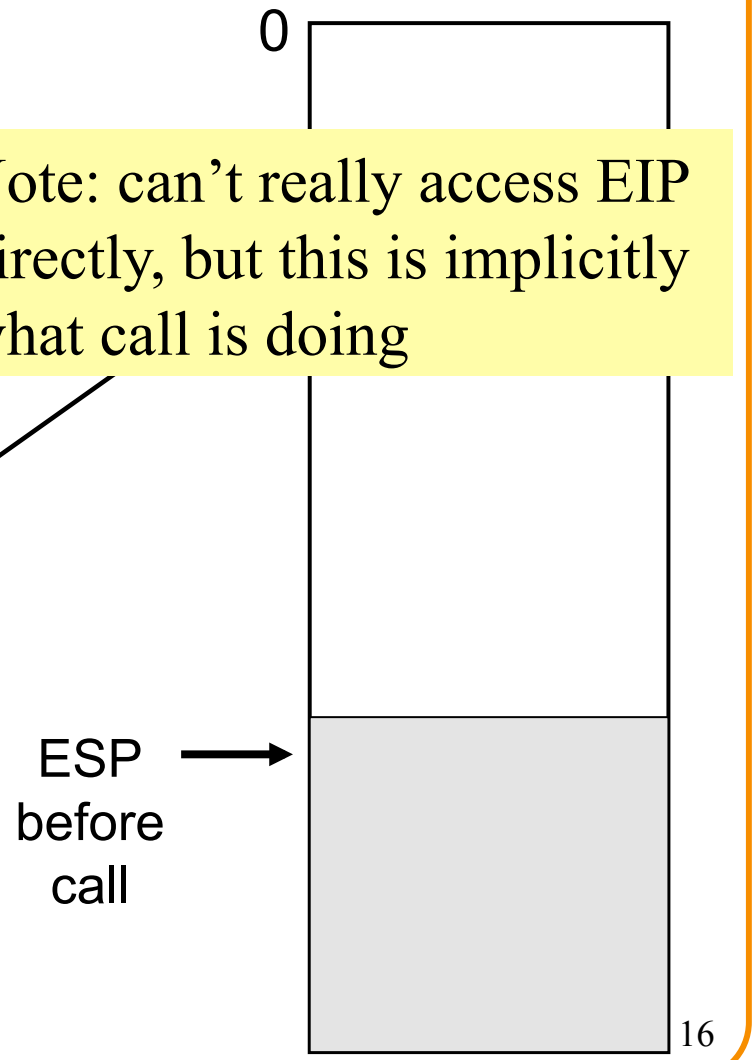


Implementation of Call

- EIP (instruction pointer register) points to next instruction to be executed

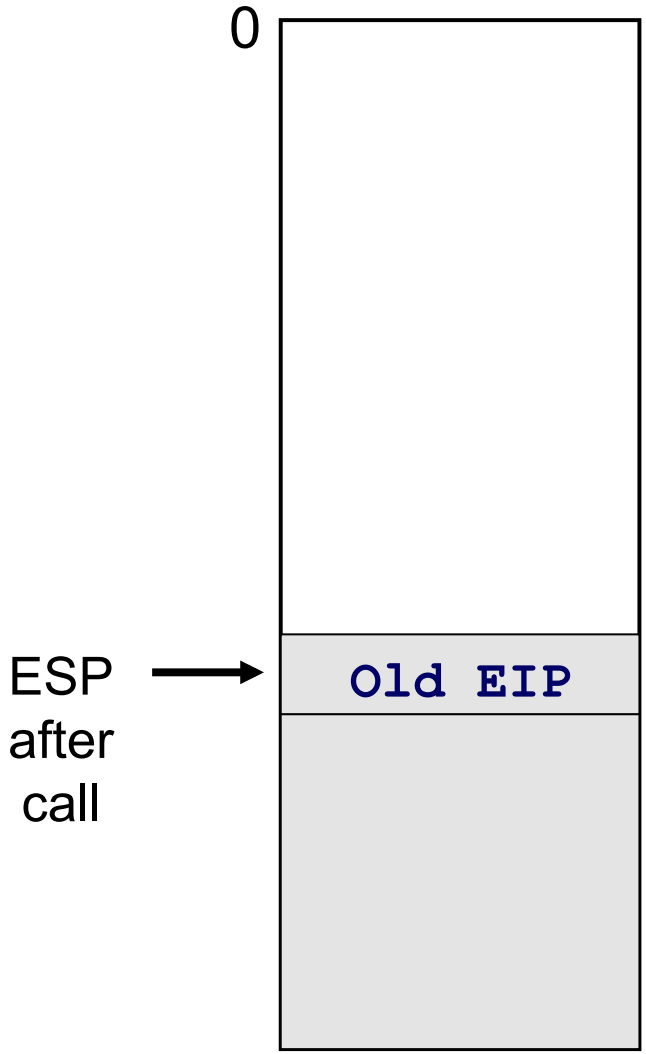
Instruction	Effective Operations
`pushl src`	`subl $4, %esp` `movl src, (%esp)`
`popl dest`	`movl (%esp), dest` `addl $4, %esp`
`call addr`	`pushl %eip` `jmp addr`

Call instruction pushes return address (old EIP) onto stack



Implementation of Call

Instruction	Effective Operations
`pushl src`	`subl $4, %esp` `movl src, (%esp)`
`popl dest`	`movl (%esp), dest` `addl $4, %esp`
`call addr`	`pushl %eip` `jmp addr`

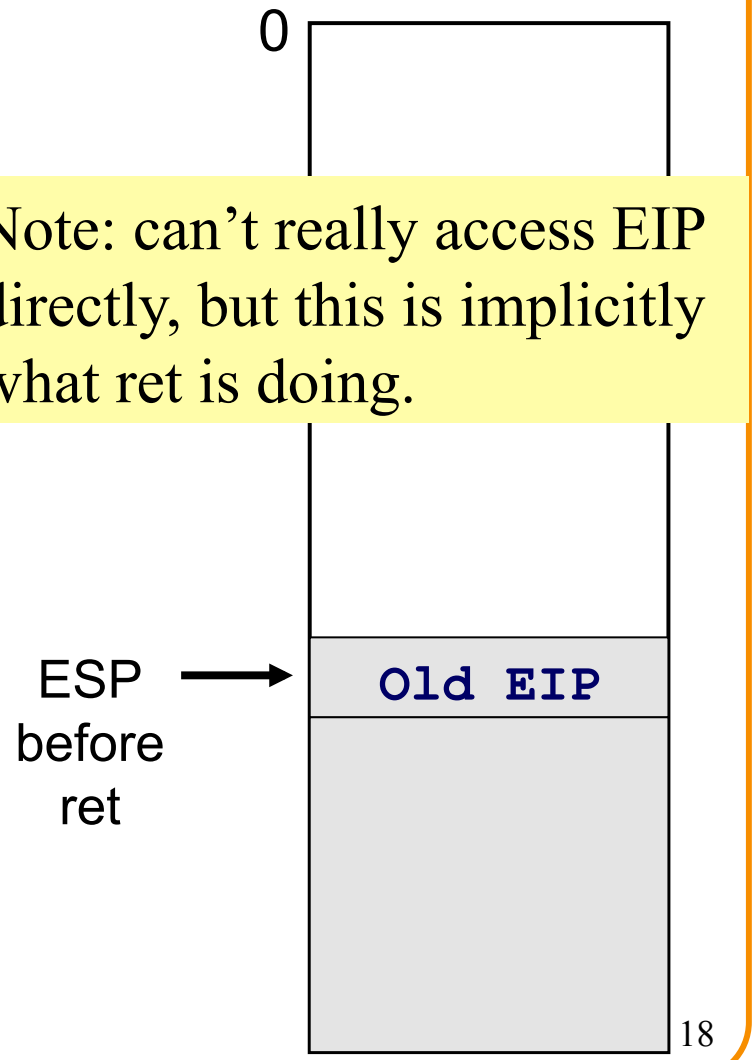


Implementation of Ret

Instruction	Effective Operations
`pushl src`	`subl $4, %esp` `movl src, (%esp)`
`popl dest`	`movl (%esp), dest` `addl $4, %esp`
`call addr`	`pushl %eip` `jmp addr`
`ret`	`pop %eip`

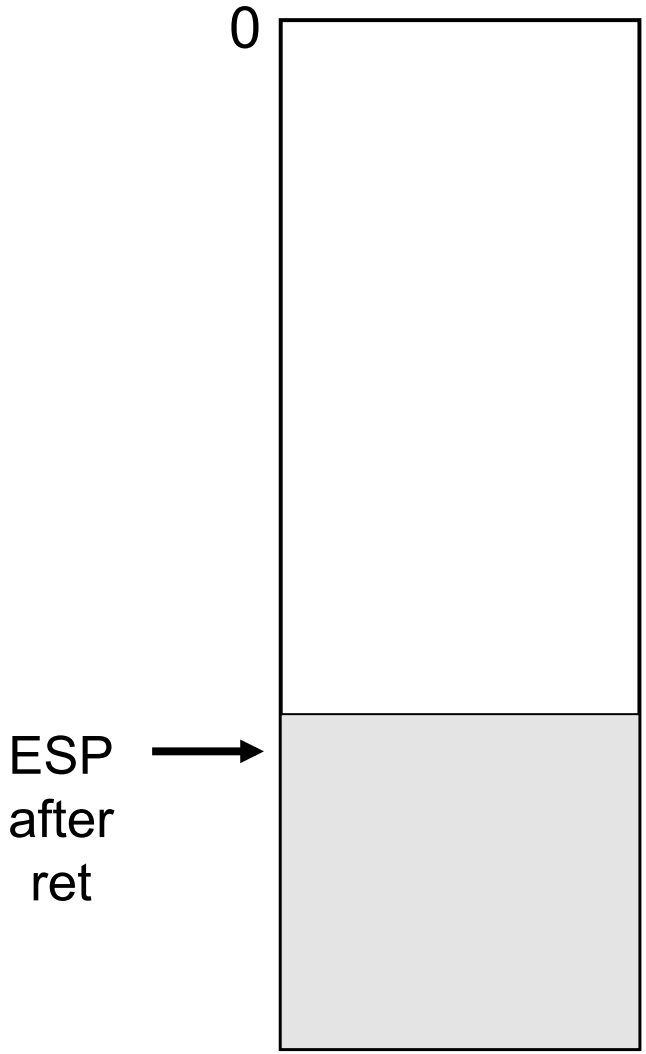
Ret instruction pops stack, thus placing return address (old EIP) into EIP

Note: can't really access EIP directly, but this is implicitly what ret is doing.



Implementation of Ret

Instruction	Effective Operations
`pushl src`	`subl $4, %esp` `movl src, (%esp)`
`popl dest`	`movl (%esp), dest` `addl $4, %esp`
`call addr`	`pushl %eip` `jmp addr`
`ret`	`pop %eip`



Problem 2: Passing Parameters

- Problem: How does caller function pass *parameters* to callee function?

```
int add3(int a, int b, int c)
{
  int d;
  d = a + b + c;
  return d;
}

int f(void)
{
  return add3(3, 4, 5);
}
```

Attempted Solution: Use Registers

- Attempted solution: Pass parameters in registers

```
f:
    movl $3, %eax
    movl $4, %ebx
    movl $5, %ecx
    call add3
    …
```

```
add3:
    …
    # Use EAX, EBX, ECX
    …
    ret
```

Attempted Solution: Use Registers

- Problem: Cannot handle nested function calls

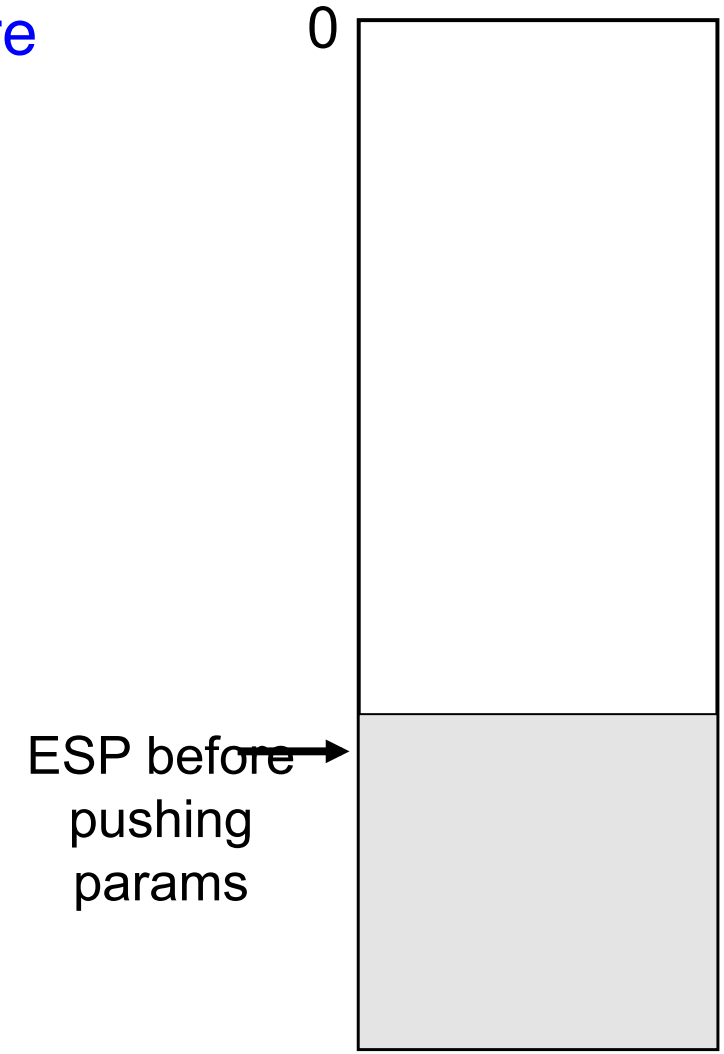
```
f:
    movl $3, %eax
    movl $4, %ebx
    movl $5, %ecx
    call add3
    …
```

```
add3:
    …
    movl $6, %eax
    call g
    # Use EAX, EBX, ECX
    # But EAX is corrupted!
    …
    ret
```

- Also: How to pass parameters that are longer than 4 bytes?

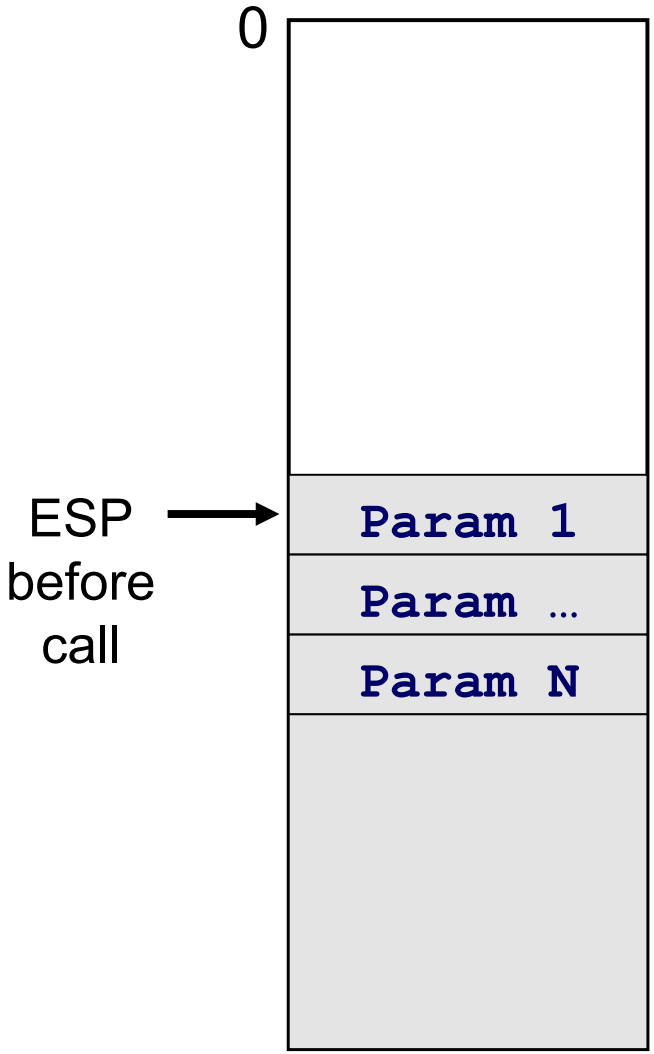
IA-32 Solution: Use the Stack

- Caller pushes parameters before executing the call instruction



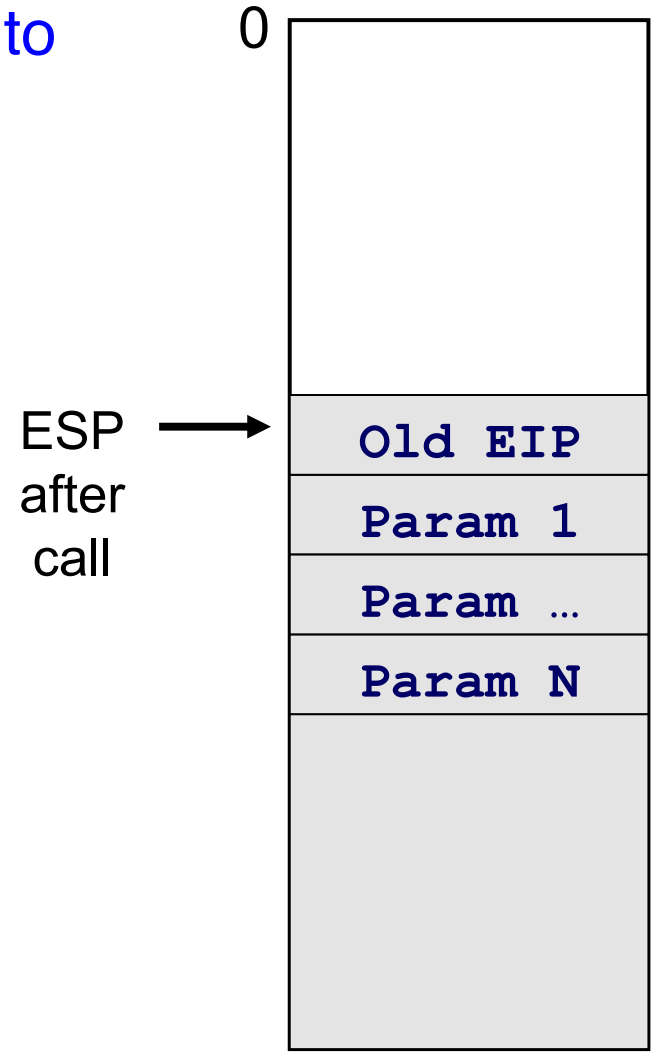
IA-32 Parameter Passing

- Caller pushes parameters in the reverse order
  - Push Nth param first
  - Push 1st param last
  - So first param is at top of the stack at the time of the Call



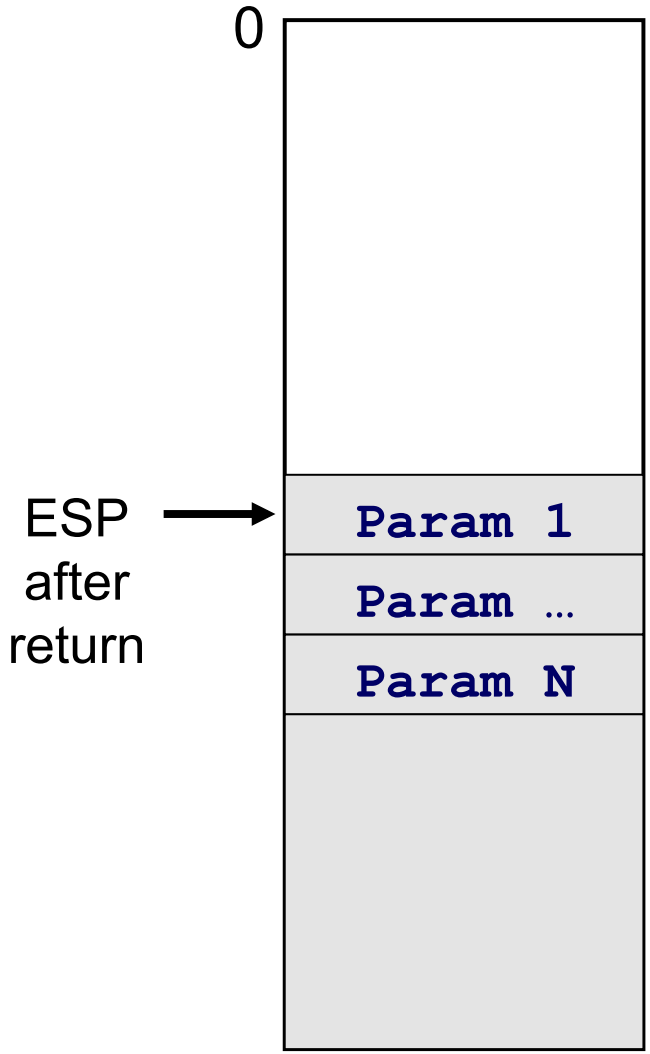
IA-32 Parameter Passing

- Callee addresses params relative to ESP: Param 1 as 4(%esp)



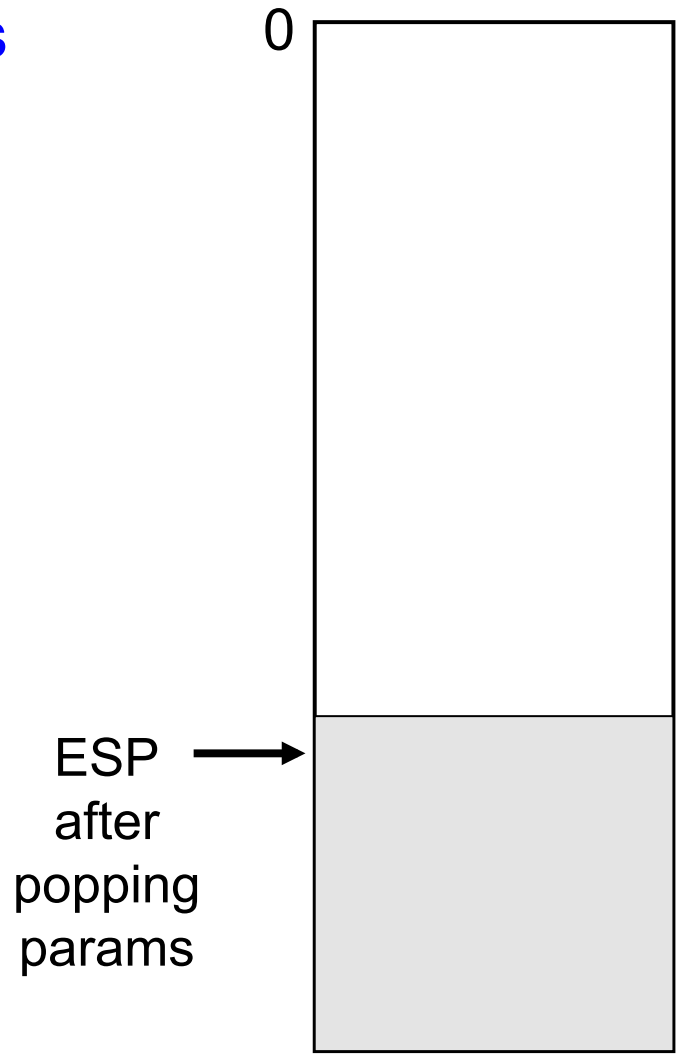
IA-32 Parameter Passing

- After returning to the caller…



IA-32 Parameter Passing

- … the caller pops the parameters from the stack



IA-32 Parameter Passing

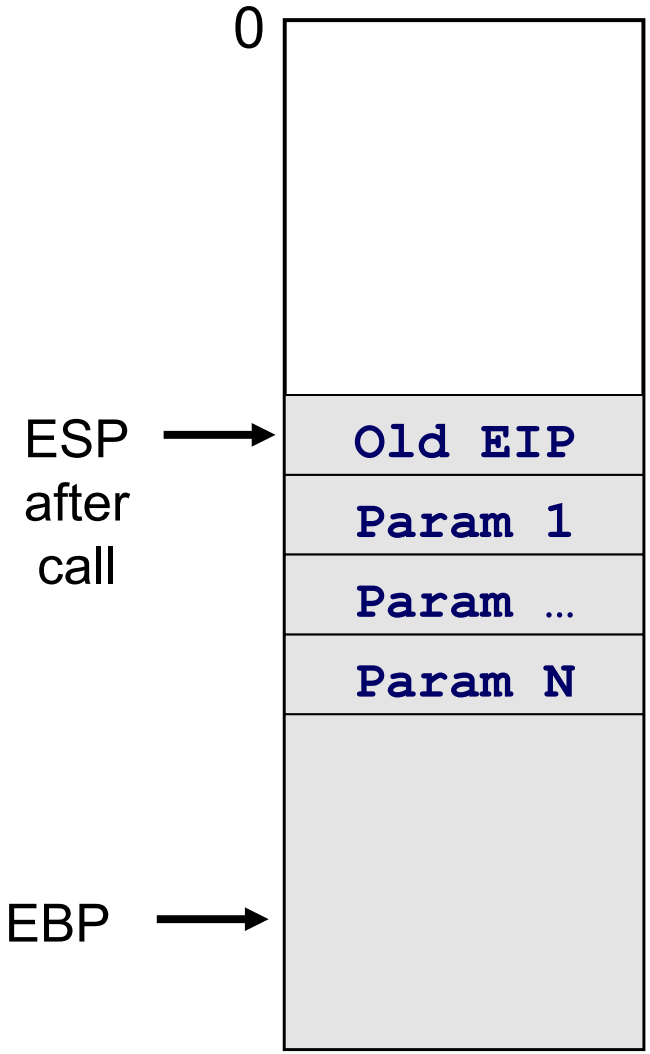
For example:

```
f:
    …
    # Push parameters
    pushl $5
    pushl $4
    pushl $3
    call add3
    # Pop parameters
    addl $12, %esp
```

```
add3:
    …
    movl 4(%esp), wherever
    movl 8(%esp), wherever
    movl 12(%esp), wherever
    …
    ret
```

Base Pointer Register: EBP

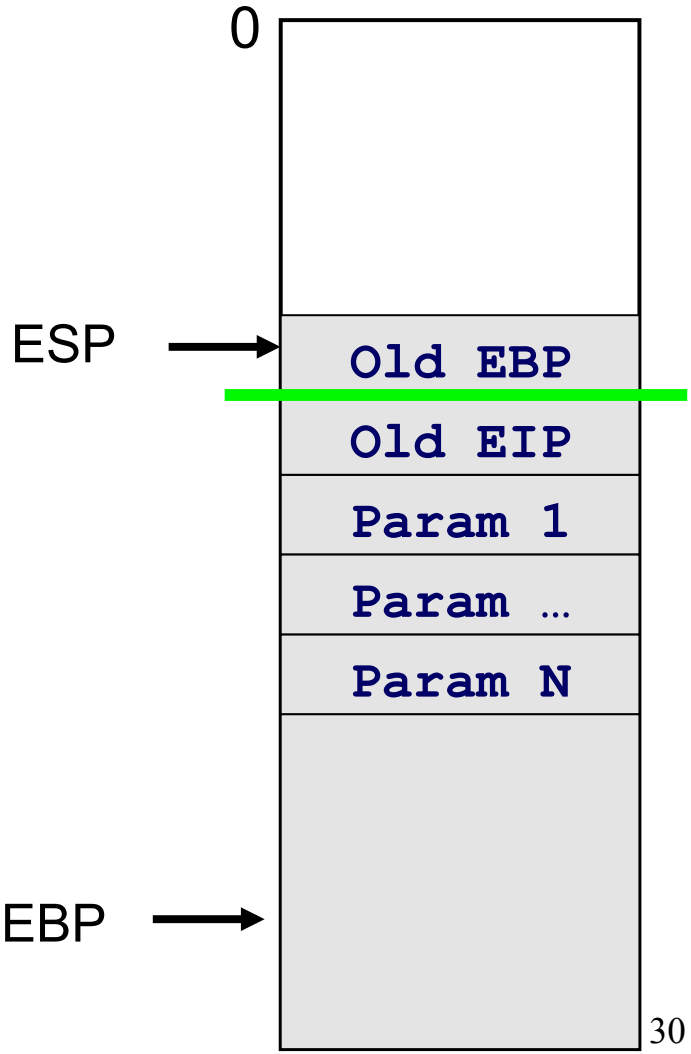
- Problem:
  - As callee executes, ESP may change
    - E.g., preparing to call another function
  - Error-prone for callee to reference params as offsets relative to ESP
- Solution:
  - Use EBP as fixed reference point to access params



Using EBP

- Need to save old value of EBP
  - Before overwriting EBP register
- Callee executes "prolog"

```
pushl %ebp
movl %esp, %ebp
```

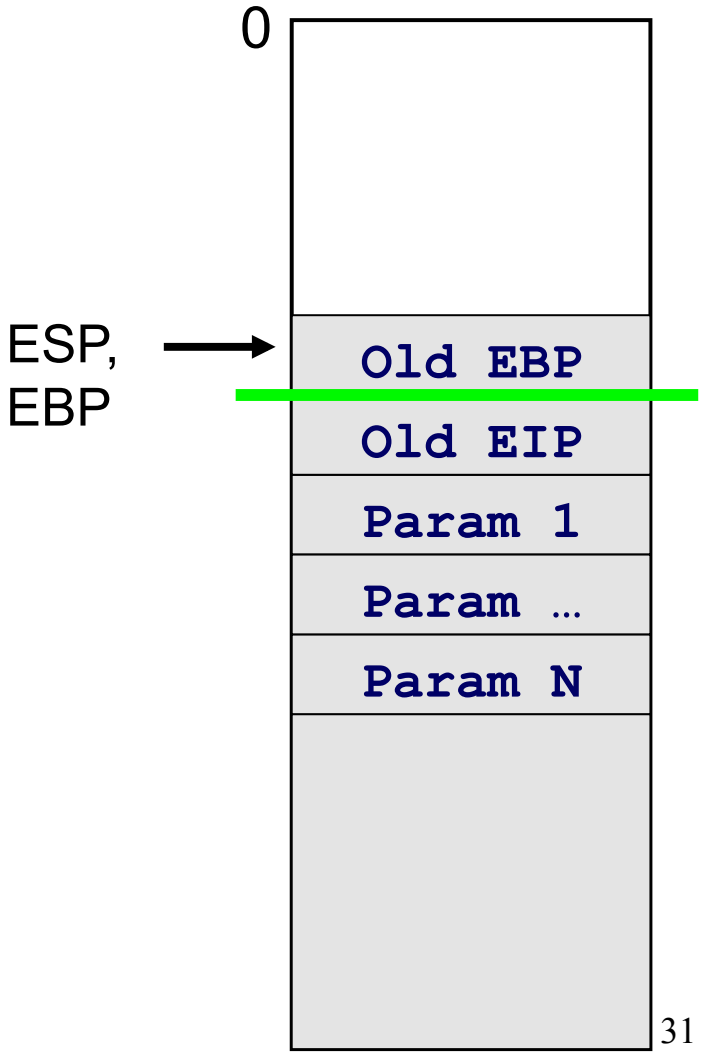


Base Pointer Register: EBP

- Callee executes “prolog”

```
pushl %ebp
movl %esp, %ebp
```

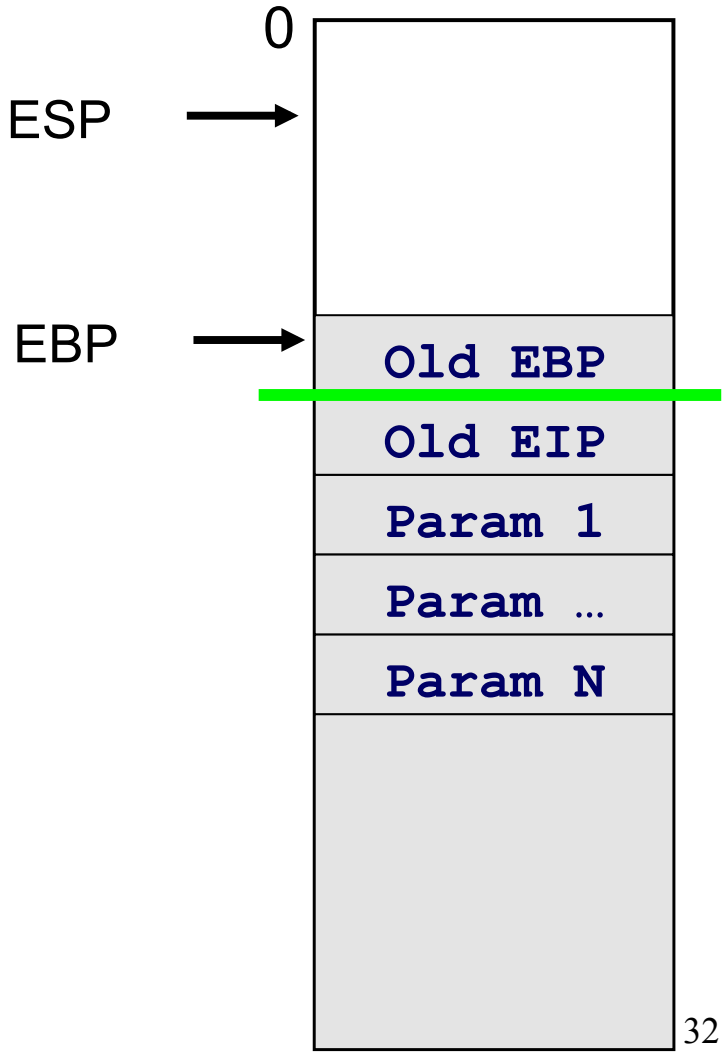
- Regardless of ESP, callee can reference param 1 as 8(%ebp), param 2 as 12(%ebp), etc.



Base Pointer Register: EBP

- Before returning, callee must restore ESP and EBP to their old values
- Callee executes "epilog"

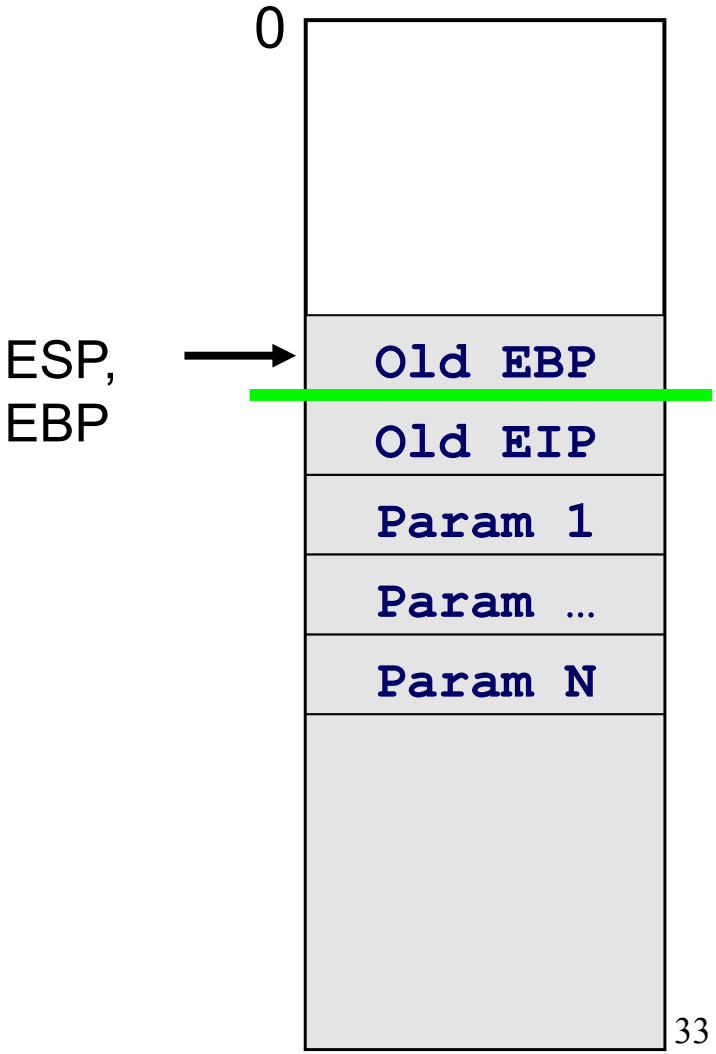
```
movl %ebp, %esp
popl %ebp
ret
```



Base Pointer Register: EBP

- Callee executes “epilog”

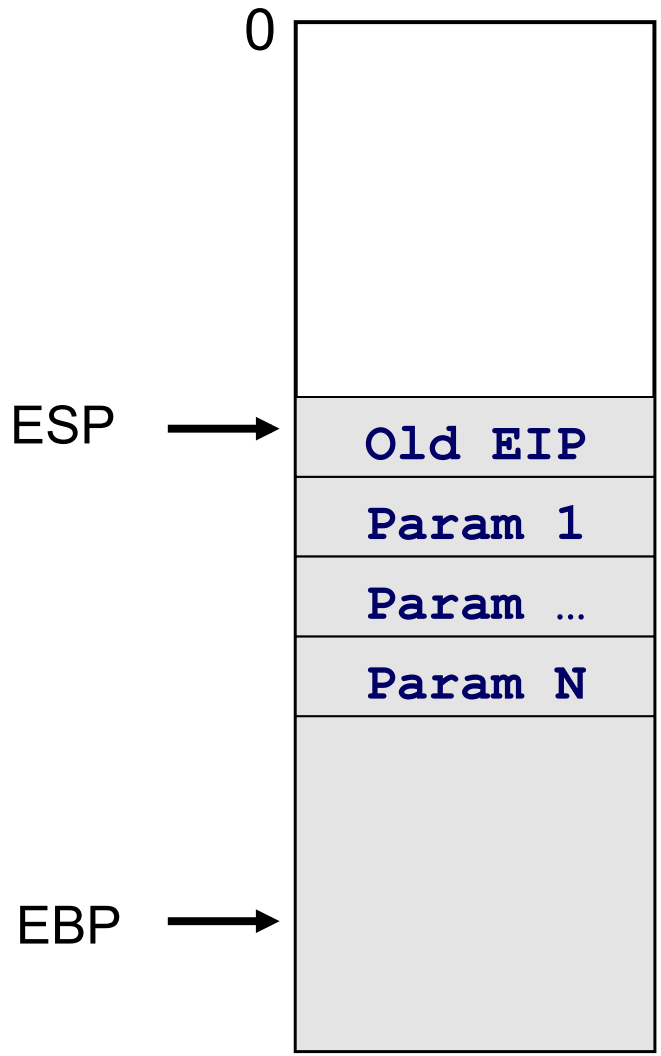
```
movl %ebp, %esp
popl %ebp
ret
```



Base Pointer Register: EBP

- Callee executes “epilog”

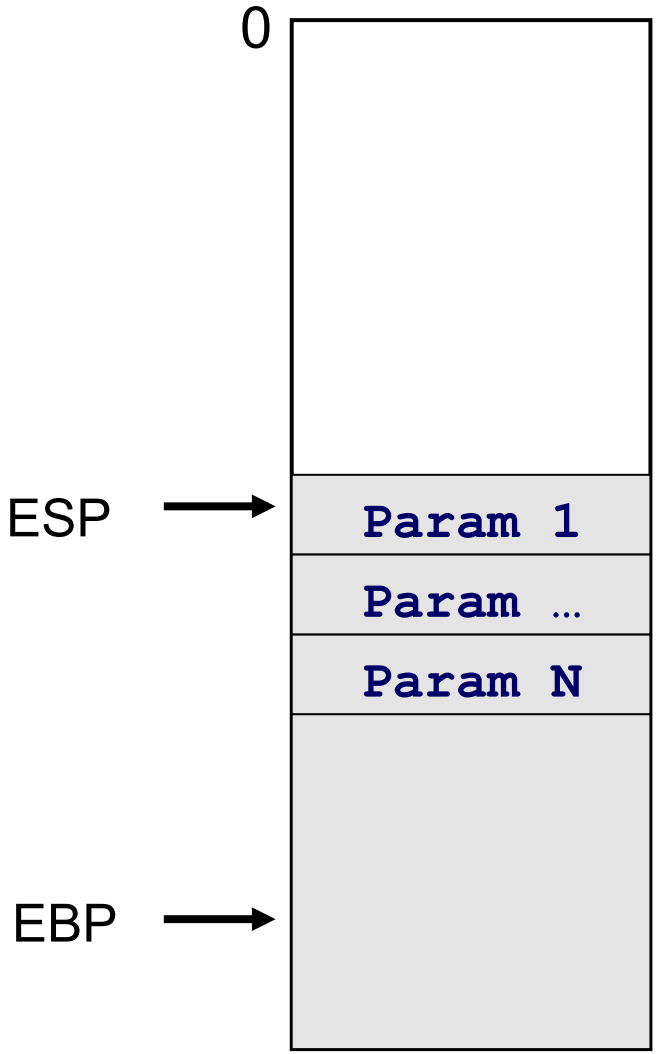
```
movl %ebp, %esp
popl %ebp
ret
```



Base Pointer Register: EBP

- Callee executes “epilog”

```
movl %ebp, %esp
popl %ebp
ret
```



Problem 3: Storing Local Variables

- Where does callee function store its *local variables?*

```
int add3(int a, int b, int c)
{
  int d;
  d = a + b + c;
  return d;
}

int foo(void)
{
  return add3(3, 4, 5);
}
```

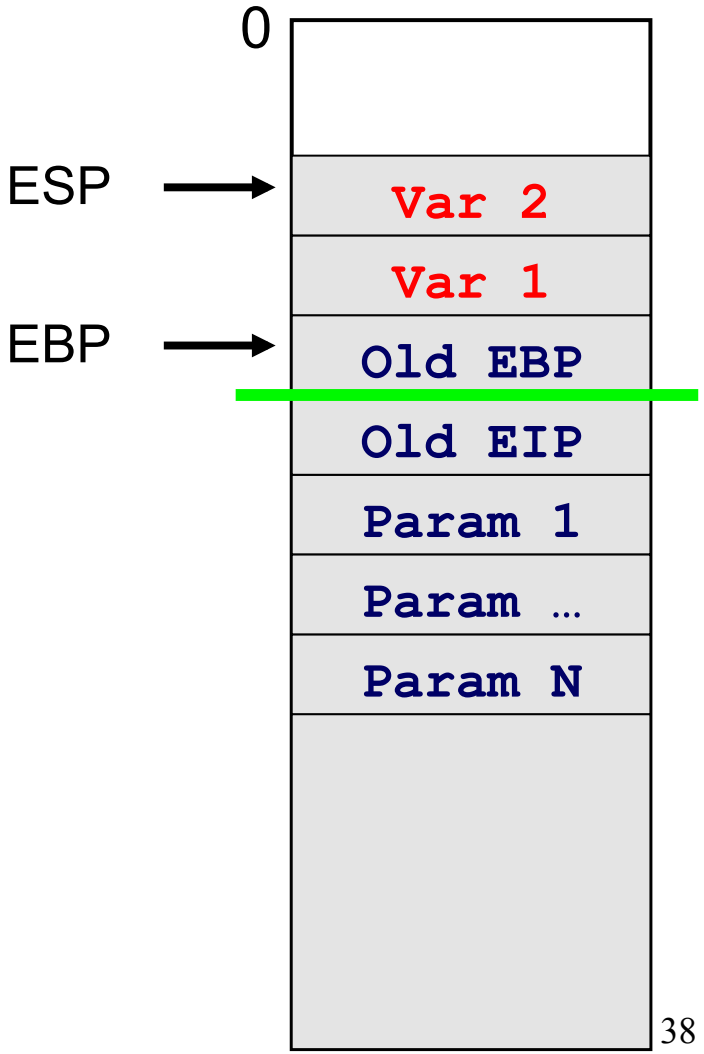
IA-32 Solution: Use the Stack

- Local variables:
  - Short-lived, so don't need a permanent location in memory
  - Size known in advance, so don't need to allocate on the heap
- So, the function just uses the top of the stack
  - Store local variables on the top of the stack
  - The local variables disappear after the function returns

```
int add3(int a, int b, int c)
{
  int d;
  d = a + b + c;
  return d;
}

int foo(void)
{
  return add3(3, 4, 5);
}
```

IA-32 Local Variables

- Local variables of the callee are allocated on the stack
- Allocation done by moving the stack pointer
- Example: allocate memory for two integers
  - subl $4, %esp
  - subl $4, %esp
  - (or equivalently, subl $8, %esp)
- Reference local variables as negative offsets relative to EBP
  - -4(%ebp)
  - -8(%ebp)



IA-32 Local Variables

For example:

```
add3:
    …
    # Allocate space for d
    subl $4, %esp
    …
    # Access d
    movl whatever, -4(%ebp)
    …
    ret
```

Problem 4: Handling Registers

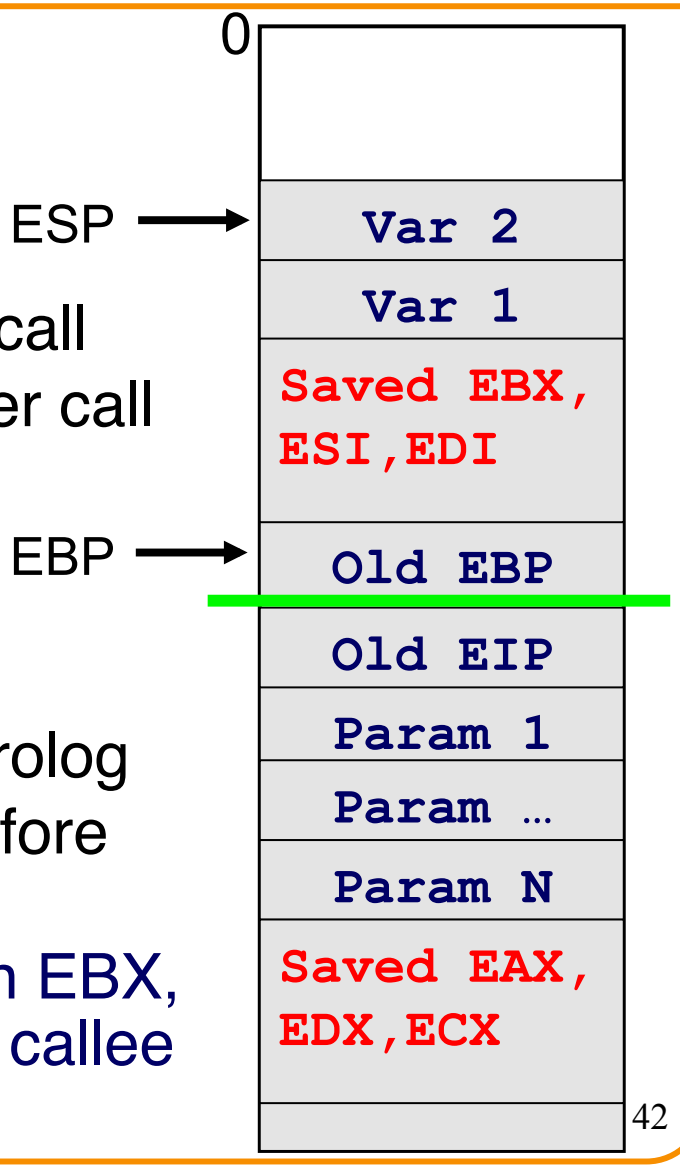
- Problem: How do caller and callee functions use *same registers* without interference?
- Registers are a finite resource!
  - In principle: Each function should have its own set of registers
  - In reality: All functions must use the same small set of registers
- Callee may use a register that the caller also is using
  - When callee returns control to caller, old register contents may be lost
  - Caller function cannot continue where it left off

IA-32 Solution: Define a Convention

- IA-32 solution: save the registers on the stack
  - Someone must save old register contents
  - Someone must later restore the register contents
- Define a convention for who saves and restores which registers

IA-32 Register Handling

- Caller-save registers
  - **`EAX, EDX, ECX`**
  - If necessary…
    - Caller saves on stack before call
    - Caller restores from stack after call
- Callee-save registers
  - **`EBX, ESI, EDI`**
  - If necessary…
    - Callee saves on stack after prolog
    - Callee restores from stack before epilog
  - Caller can assume that values in EBX, ESI, EDI will not be changed by callee



Problem 5: Return Values

- Problem: How does callee function send return value back to caller function?
- In principle:
  - Store return value in stack frame of caller
- Or, for efficiency:
  - Known small size => store return value in register
  - Other => store return value in stack

```
int add3(int a, int b, int c)
{
  int d;
  d = a + b + c;
  return d;
}

int foo(void)
{
  return add3(3, 4, 5);
}
```

IA-32 Return Values

IA-32 Convention:

- Integral type or pointer:
  - Store return value in EAX
  - char, short, int, long, pointer
- Floating-point type:
  - Store return value in floating-point register
  - (Beyond scope of course)
- Structure:
  - Store return value on stack
  - (Beyond scope of course)

```
int add3(int a, int b, int c)
{
  int d;
  d = a + b + c;
  return d;
}

int foo(void)
{
  return add3(3, 4, 5);
}
```

Stack Frames

Summary of IA-32 function handling:

- Stack has one **stack frame** per active function invocation
- ESP points to top (low memory) of current stack frame
- EBP points to bottom (high memory) of current stack frame
- Stack frame contains:
  - Return address (Old EIP)
  - Old EBP
  - Saved register values
  - Local variables
  - Parameters to be passed to callee function

A Simple Example

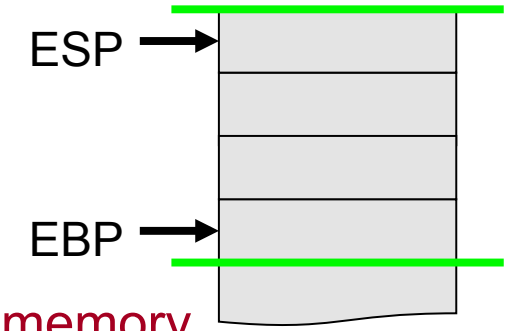
```
int add3(int a, int b, int c)
{
  int d;
  d = a + b + c;
  return d;
}
```

```
/* In some calling function */

  …
  x = add3(3, 4, 5);
  …
```

Trace of a Simple Example 1

```
x = add3(3, 4, 5);
```

Low memory



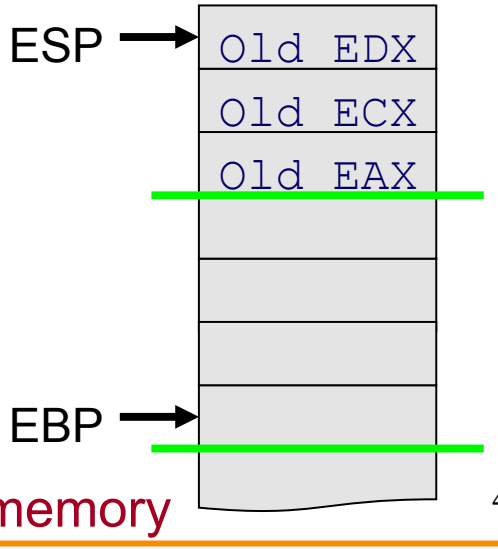
High memory

Trace of a Simple Example 2

```
x = add3(3, 4, 5);
```

Low memory

 # Save caller-save registers if necessary
pushl %eax
pushl %ecx
pushl %edx

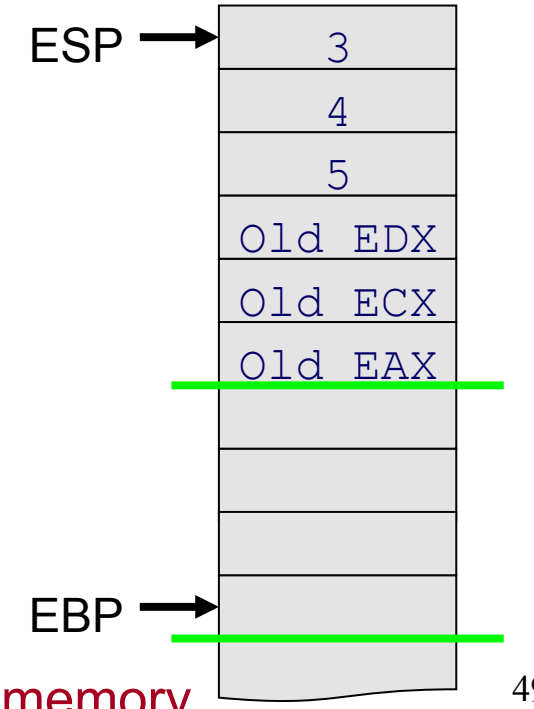


Trace of a Simple Example 3

```
x = add3(3, 4, 5);
```

Low memory

```
# Save caller-save registers if necessary
pushl %eax
pushl %ecx
pushl %edx
# Push parameters
pushl $5
pushl $4
pushl $3
```

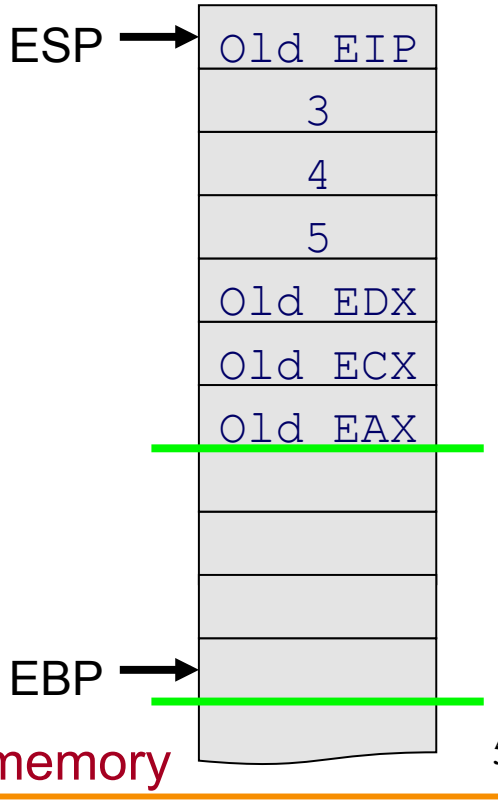


High memory

Trace of a Simple Example 4

```
x = add3(3, 4, 5);
```

```
# Save caller-save registers if necessary
pushl %eax
pushl %ecx
pushl %edx
# Push parameters
pushl $5
pushl $4
pushl $3
# Call add3
call add3
```



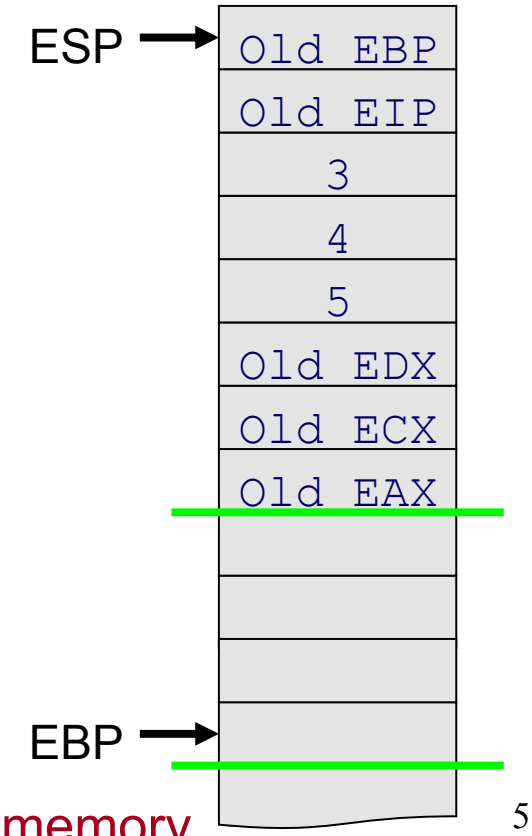
Trace of a Simple Example 5

```
int add3(int a, int b, int c) {
  int d;
  d = a + b + c;
  return d;
}
```

 # Save old EBP
pushl %ebp

Prolog

Low memory



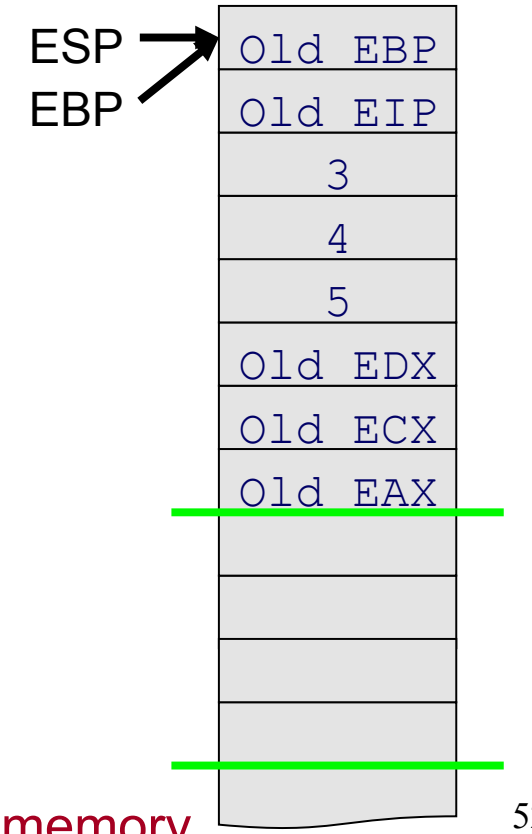
Trace of a Simple Example 6

```
int add3(int a, int b, int c) {
  int d;
  d = a + b + c;
  return d;
}
```

```
# Save old EBP
pushl %ebp
# Change EBP
movl %esp, %ebp
```

Prolog

Low memory



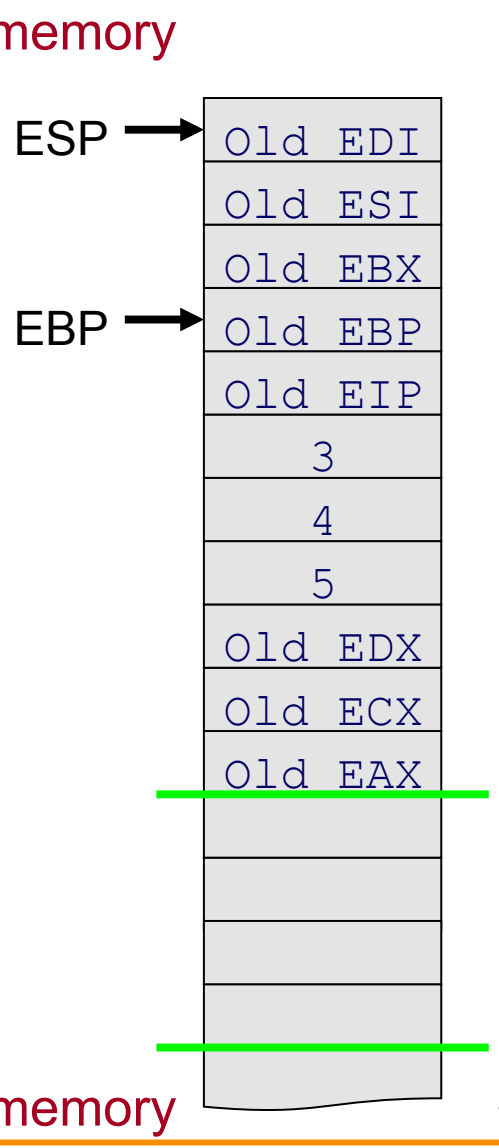
High memory

Trace of a Simple Example 7

```
int add3(int a, int b, int c) {
  int d;
  d = a + b + c;
  return d;
}
```

```
# Save old EBP
pushl %ebp
# Change EBP
movl %esp, %ebp
# Save caller-save registers if necessary
pushl %ebx
pushl %esi
pushl %edi
```

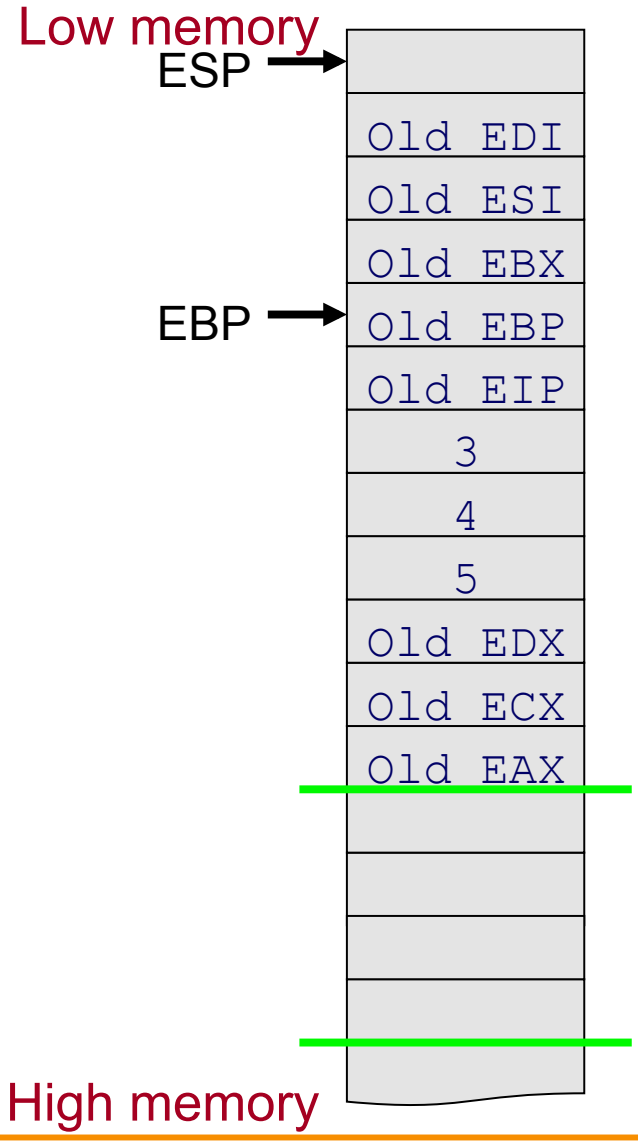
Unnecessary here; add3 will not change the values in these registers



Trace of a Simple Example 8

```
int add3(int a, int b, int c) {
  int d;
  d = a + b + c;
  return d;
}
```

```
# Save old EBP
pushl %ebp
# Change EBP
movl %esp, %ebp
# Save caller-save registers if necessary
pushl %ebx
pushl %esi
pushl %edi
# Allocate space for local variable
subl $4, %esp
```



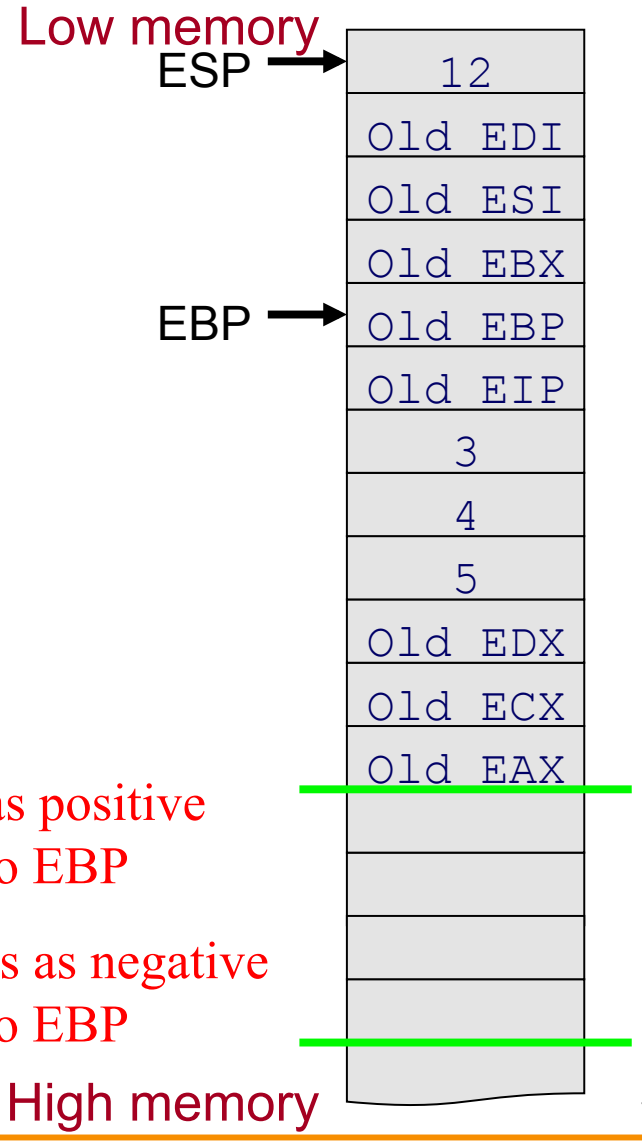
Trace of a Simple Example 9

```
int add3(int a, int b, int c) {
  int d;
  d = a + b + c;
  return d;
}
```

```
# Save old EBP
pushl %ebp
# Change EBP
movl %esp, %ebp
# Save caller-save registers if necessary
pushl %ebx
pushl %esi
pushl %edi
# Allocate space for local variable
subl $4, %esp
# Perform the addition
movl 8(%ebp), %eax
addl 12(%ebp), %eax
addl 16(%ebp), %eax
movl %eax, -16(%ebp)
```

Access params as positive offsets relative to EBP

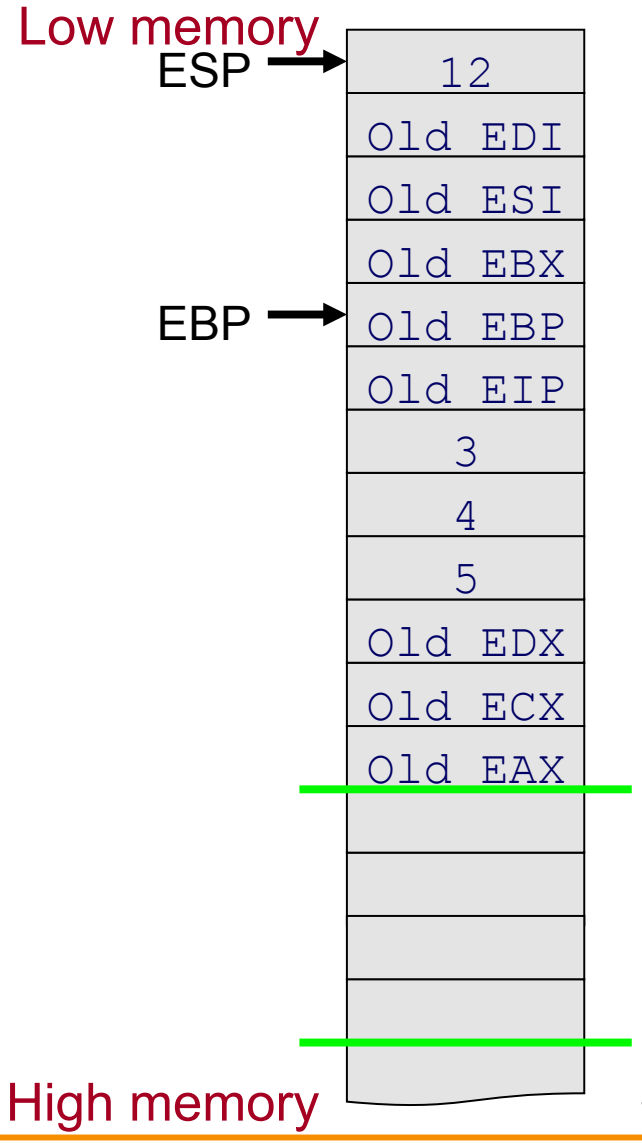
Access local vars as negative offsets relative to EBP



Trace of a Simple Example 10

```
int add3(int a, int b, int c) {
  int d;
  d = a + b + c;
  return d;
}
```

```
# Copy the return value to EAX
movl -16(%ebp), %eax
# Restore callee-save registers if necessary
movl -12(%ebp), %edi
movl -8(%ebp), %esi
movl -4(%ebp), %ebx
```

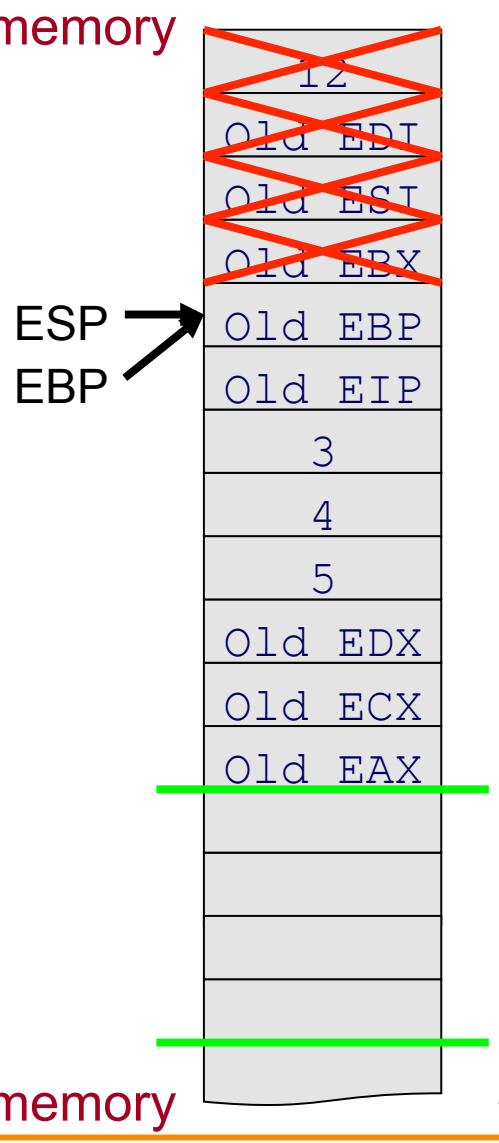


Trace of a Simple Example 11

```
int add3(int a, int b, int c) {
  int d;
  d = a + b + c;
  return d;
}
```

```
# Copy the return value to EAX
movl -16(%ebp), %eax
# Restore callee-save registers if necessary
movl -12(%ebp), %edi
movl -8(%ebp), %esi
movl -4(%ebp), %ebx
# Restore ESP
movl %ebp, %esp
```

Epilog

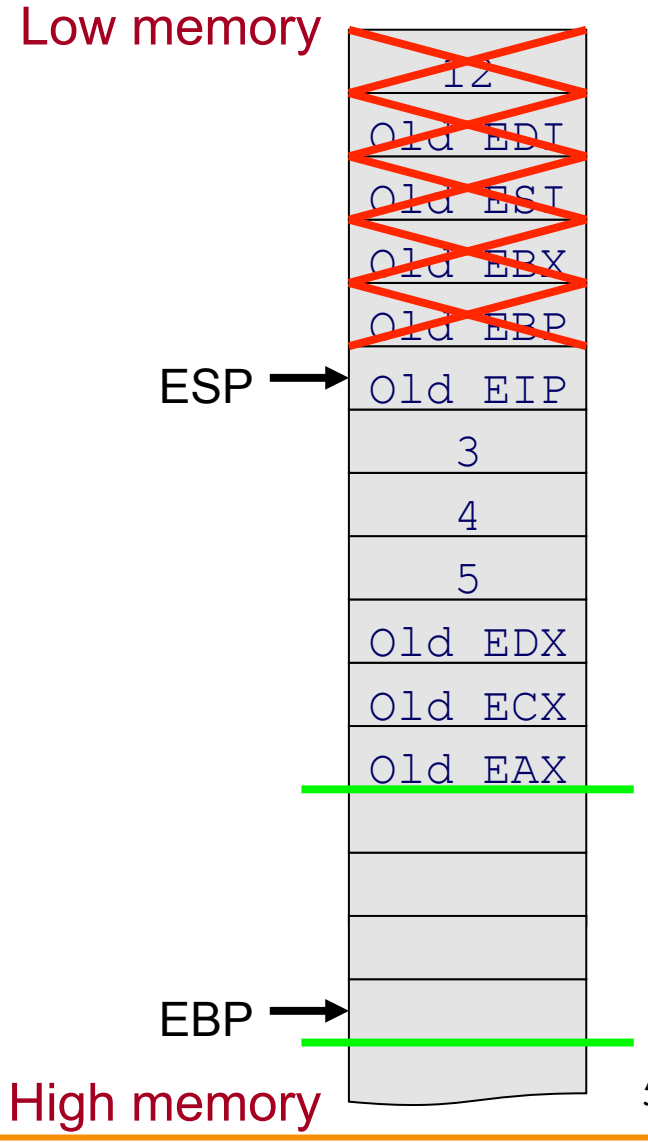


Trace of a Simple Example 12

```
int add3(int a, int b, int c) {
  int d;
  d = a + b + c;
  return d;
}
```

```
# Copy the return value to EAX
movl -16(%ebp), %eax
# Restore callee-save registers if necessary
movl -12(%ebp), %edi
movl -8(%ebp), %esi
movl -4(%ebp), %ebx
# Restore ESP
movl %ebp, %esp
# Restore EBP
popl %ebp
```

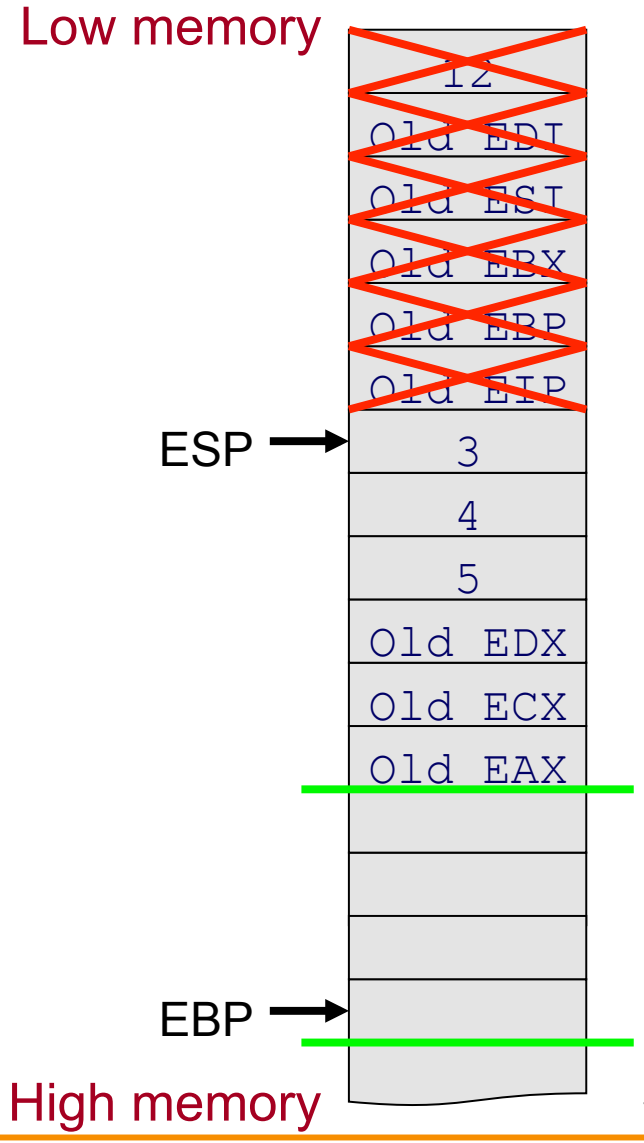
Epilog



Trace of a Simple Example 13

```
int add3(int a, int b, int c) {
  int d;
  d = a + b + c;
  return d;
}
```

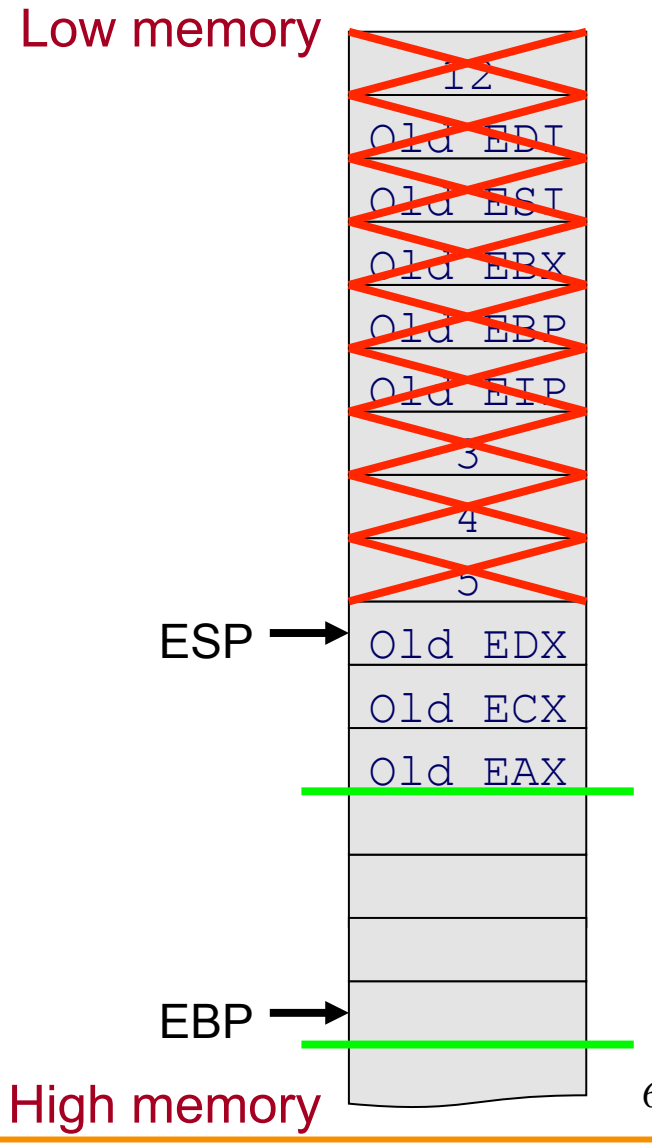
```
# Copy the return value to EAX
movl -16(%ebp), %eax
# Restore callee-save registers if necessary
movl -12(%ebp), %edi
movl -8(%ebp), %esi
movl -4(%ebp), %ebx
# Restore ESP
movl %ebp, %esp
# Restore EBP
popl %ebp
# Return to calling function
ret
```



Trace of a Simple Example 14

```
x = add3(3, 4, 5);
```

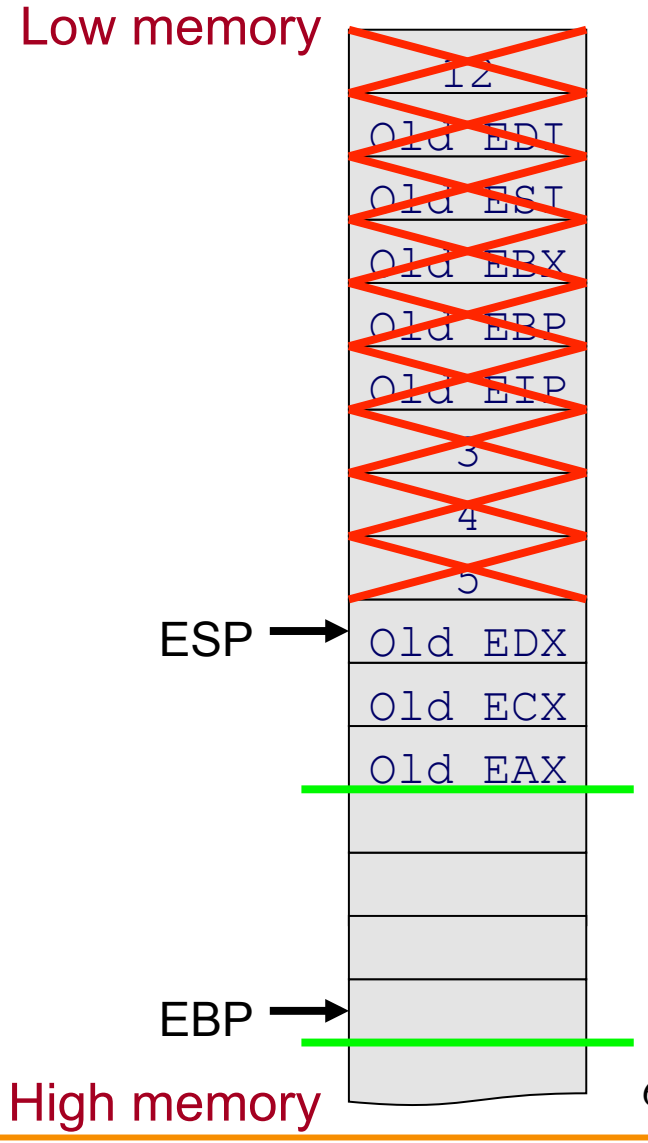
```
# Save caller-save registers if necessary
pushl %eax
pushl %ecx
pushl %edx
# Push parameters
pushl $5
pushl $4
pushl $3
# Call add3
call add3
# Pop parameters
addl $12, %esp
```



Trace of a Simple Example 15

```
x = add3(3, 4, 5);
```

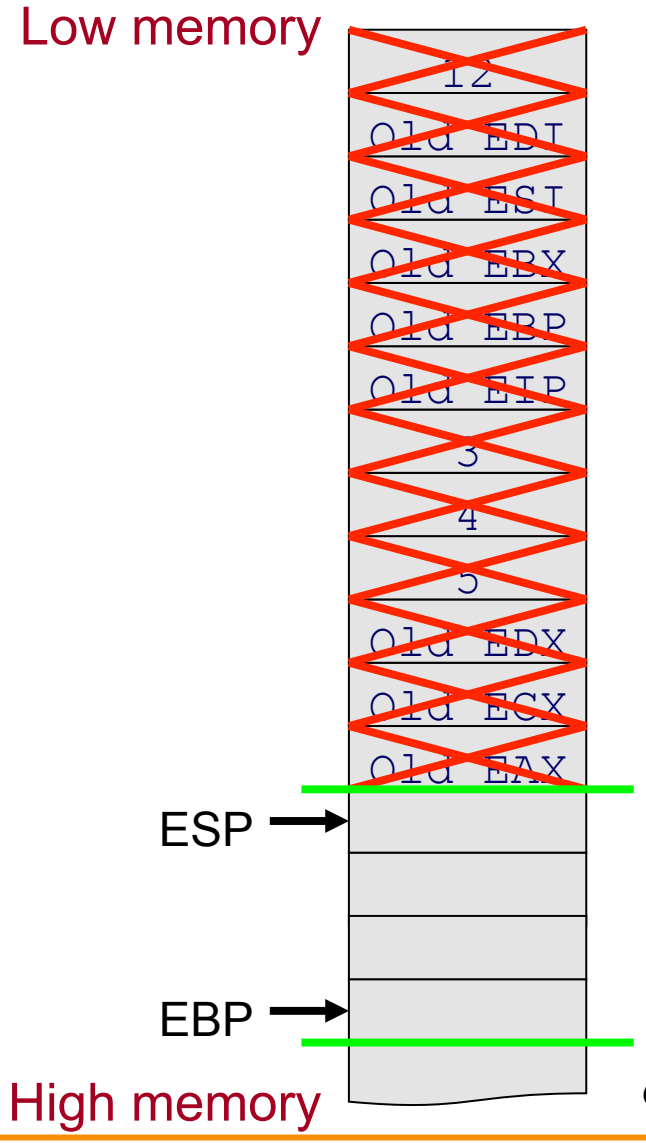
```
# Save caller-save registers if necessary
pushl %eax
pushl %ecx
pushl %edx
# Push parameters
pushl $5
pushl $4
pushl $3
# Call add3
call add3
# Pop parameters
addl %12, %esp
# Save return value
movl %eax, wherever
```



Trace of a Simple Example 16

```
x = add3(3, 4, 5);
```

```
# Save caller-save registers if necessary
pushl %eax
pushl %ecx
pushl %edx
# Push parameters
pushl $5
pushl $4
pushl $3
# Call add3
call add3
# Pop parameters
addl %12, %esp
# Save return value
movl %eax, wherever
# Restore caller-save registers if necessary
popl %edx
popl %ecx
popl %eax
```

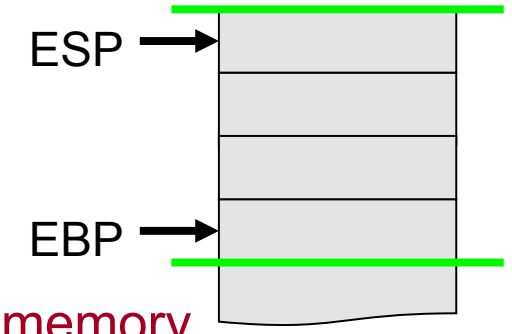


Trace of a Simple Example 17

```
x = add3(3, 4, 5);
```

Low memory

```
# Save caller-save registers if necessary
pushl %eax
pushl %ecx
pushl %edx
# Push parameters
pushl $5
pushl $4
pushl $3
# Call add3
call add3
# Pop parameters
addl %12, %esp
# Save return value
movl %eax, wherever
# Restore caller-save registers if necessary
popl %edx
popl %ecx
popl %eax
# Proceed!
…
```



High memory

Summary

- Calling and returning
  - Call instruction: push EIP onto stack and jump
  - Ret instruction: pop stack to EIP
- Passing parameters
  - Caller pushes onto stack
  - Callee accesses as positive offsets from EBP
  - Caller pops from stack

Summary (cont.)

- Storing local variables
  - Callee pushes on stack
  - Callee accesses as negative offsets from EBP
  - Callee pops from stack
- Handling registers
  - Caller saves and restores EAX, ECX, EDX if necessary
  - Callee saves and restores EBX, ESI, EDI if necessary
- Returning values
  - Callee returns data of integral types and pointers in EAX

Assemblers and Linkers

Jennifer Rexford

Goals for this Lecture

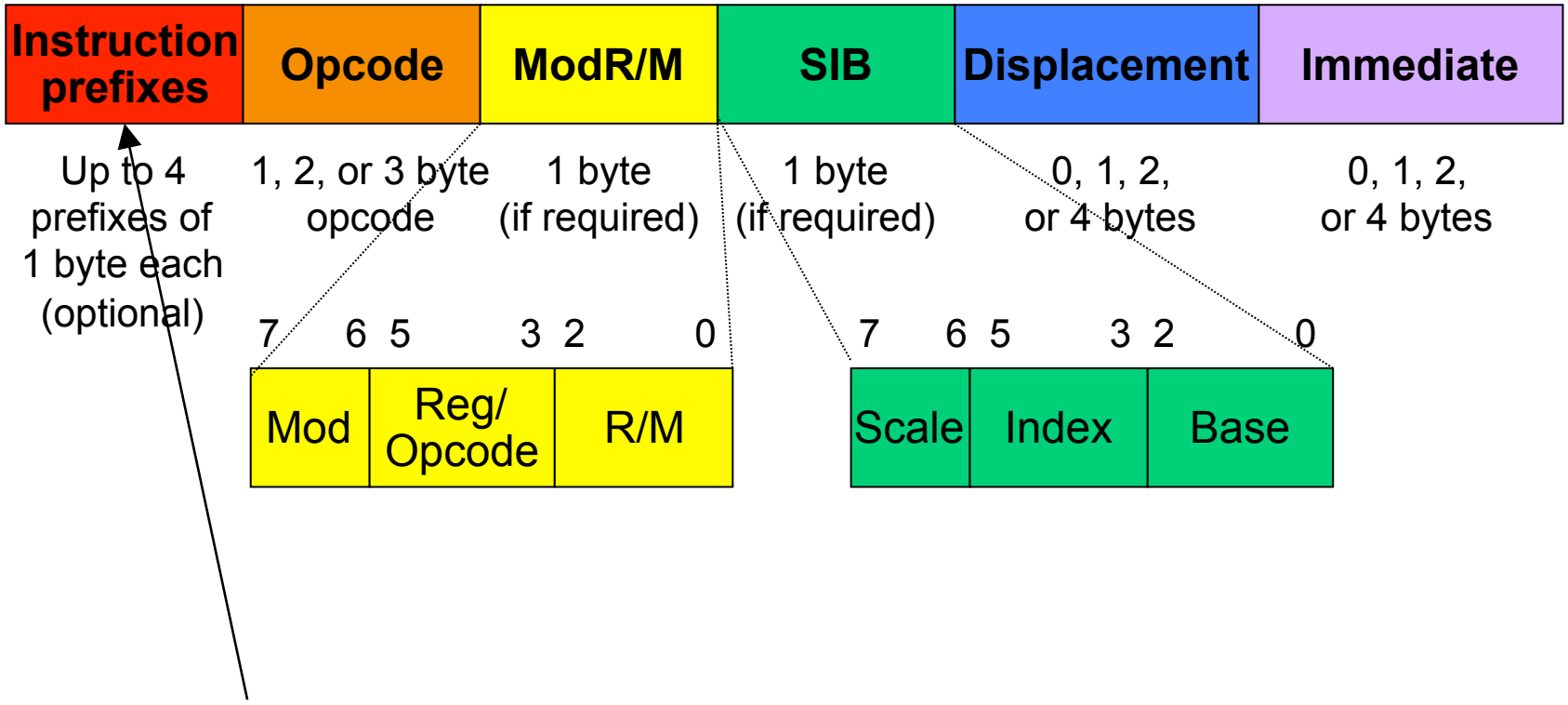
- Machine language
  - The last stop on the "language levels" tour
- IA-32 machine language
  - Useful for the next programming assignment
- Assembly and linking processes
  - To understand how to create an executable

Part 1: Machine Language

IA-32 Machine Language

- IA-32 machine language
  - Difficult to generalize about IA-32 instruction format
  - Generally, instructions use the following format
- We'll go over
  - The format of instructions
  - Two example instructions
- Just to give a sense of how it works…

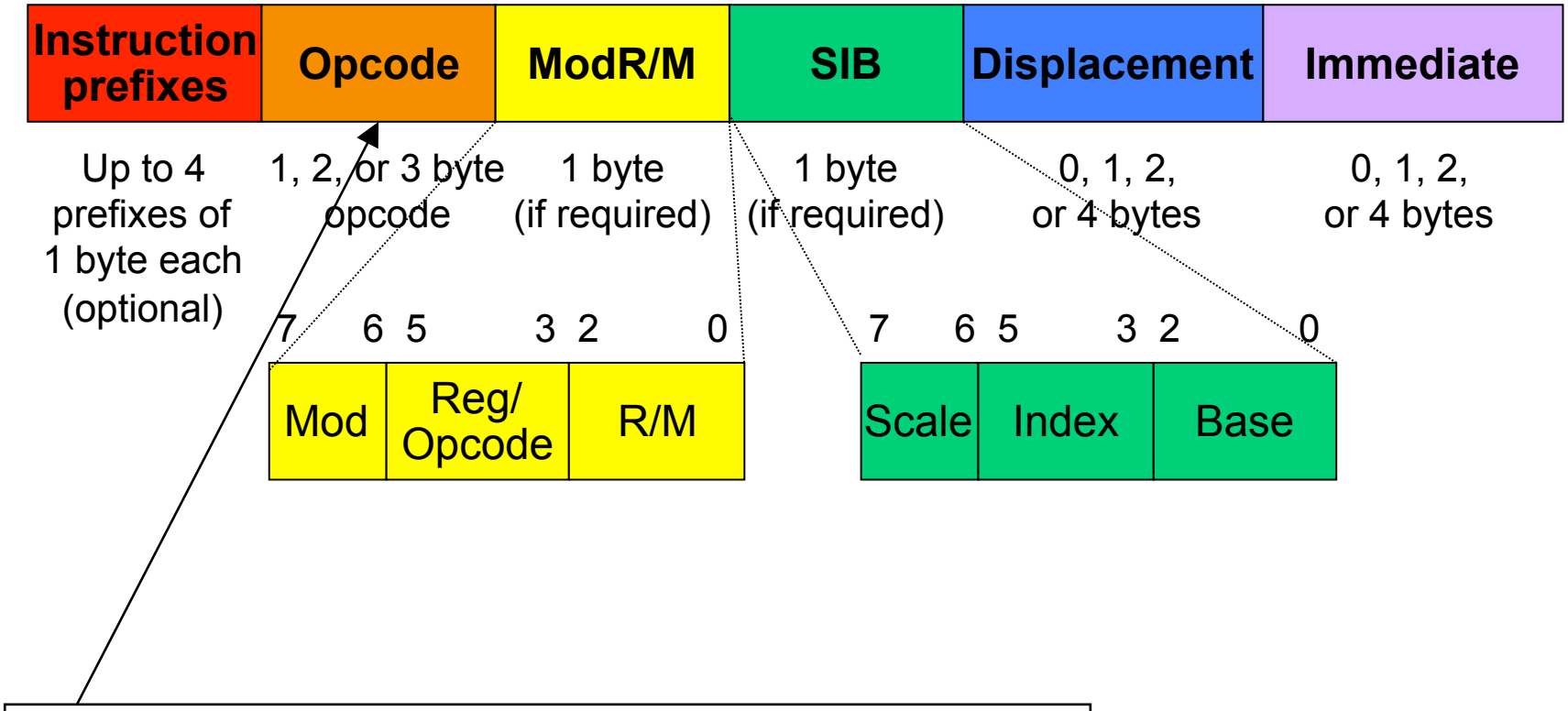
IA-32 Instruction Format



Instruction prefix

- Sometimes a repeat count
- Rarely used; don't be concerned

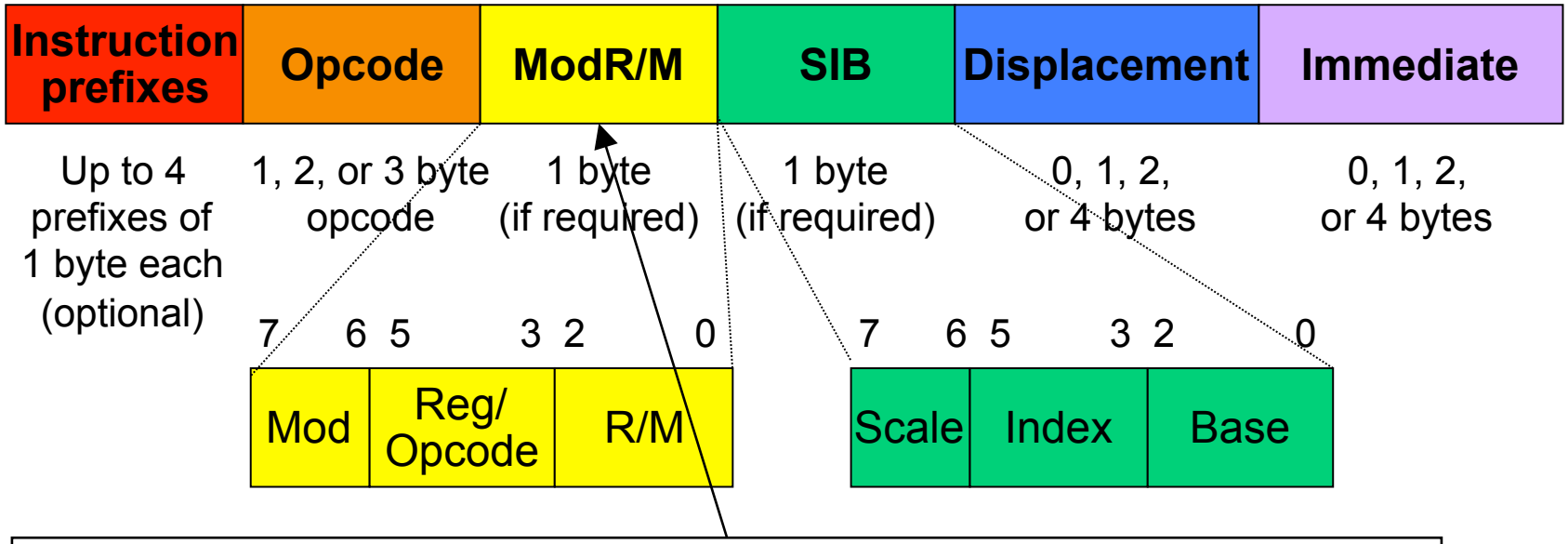
IA-32 Instruction Format (cont.)



Opcode

- Specifies which operation should be performed
- Add, move, call, etc.

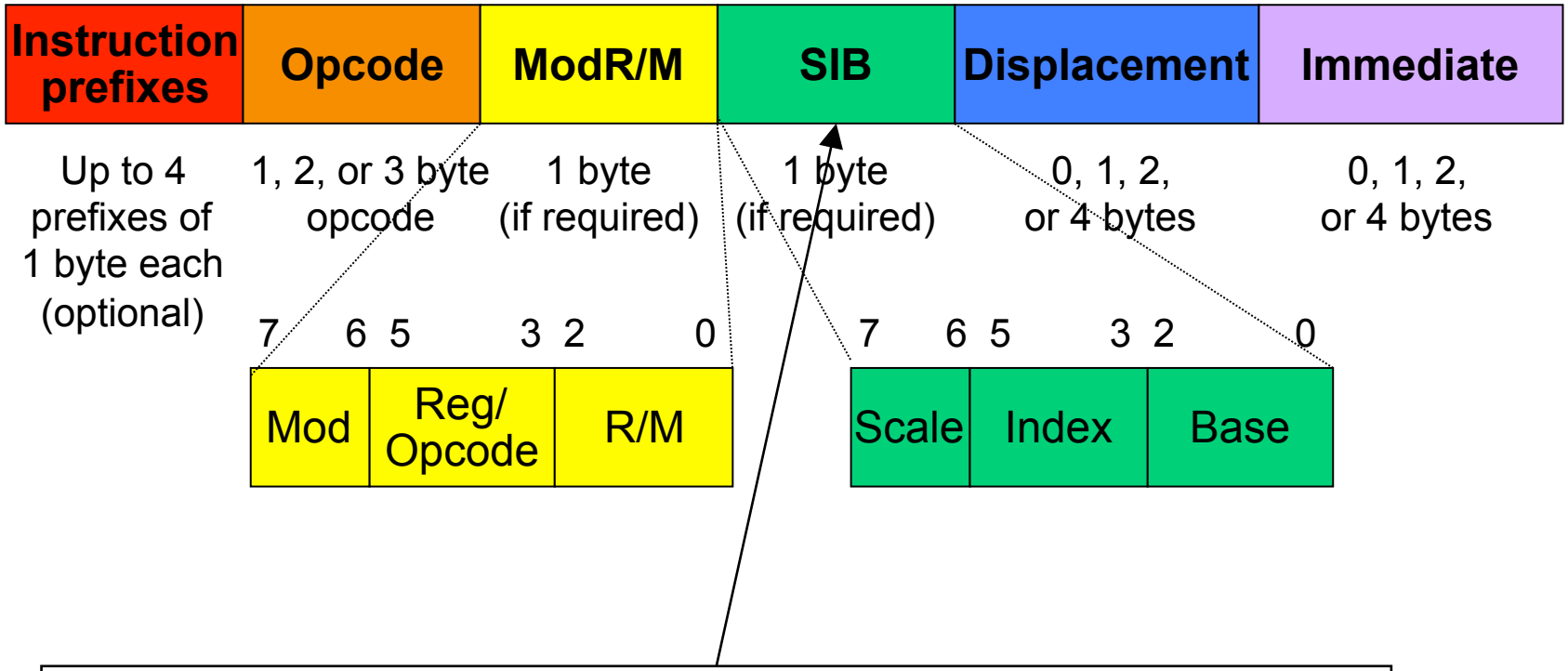
IA-32 Instruction Format (cont.)



ModR/M

- Specifies types of operands (immediate, register, memory)
- Specifies sizes of operands (byte, word, long)
- Sometimes denotes a register:
  000 = EAX/AL; 011 = EBX/BL; 001 = ECX/CL; 010 = EDX/DL;
  110 = ESI/DH; 111 = EDI/BH; 101 = EBP/CH; 110 = ESP/AH
- Sometimes contains an extension of the opcode

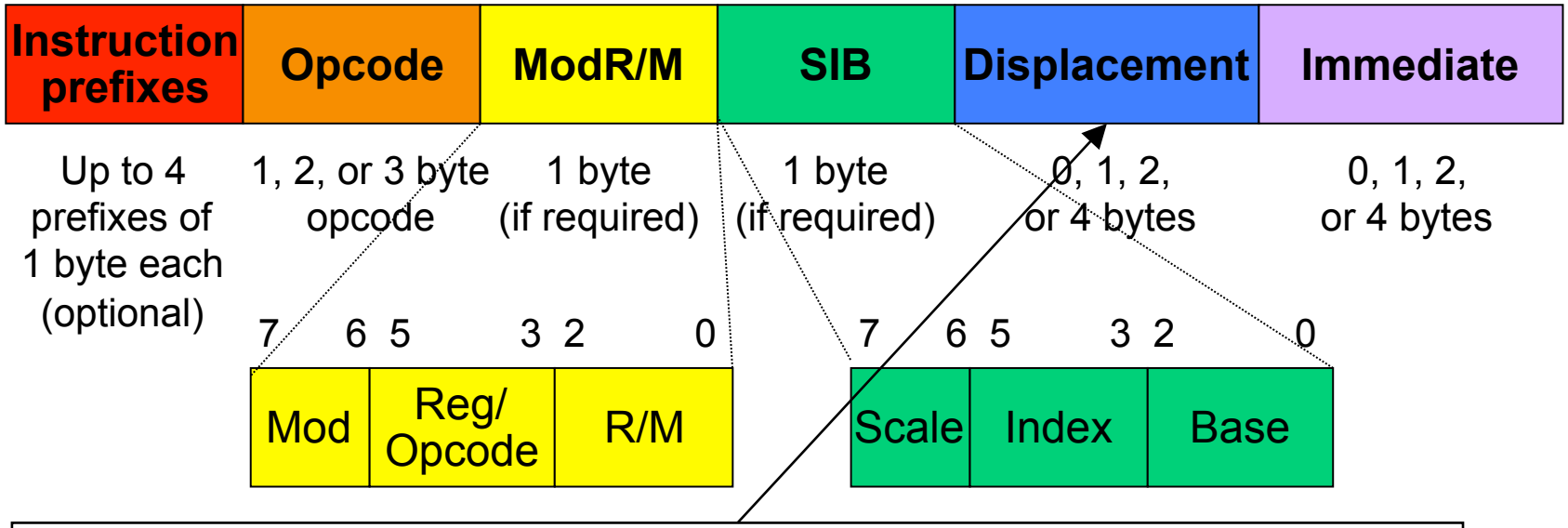
IA-32 Instruction Format (cont.)



SIB

- Used when one of the operands is a memory operand that uses a **s**cale, an **i**ndex register, and/or a **b**ase register

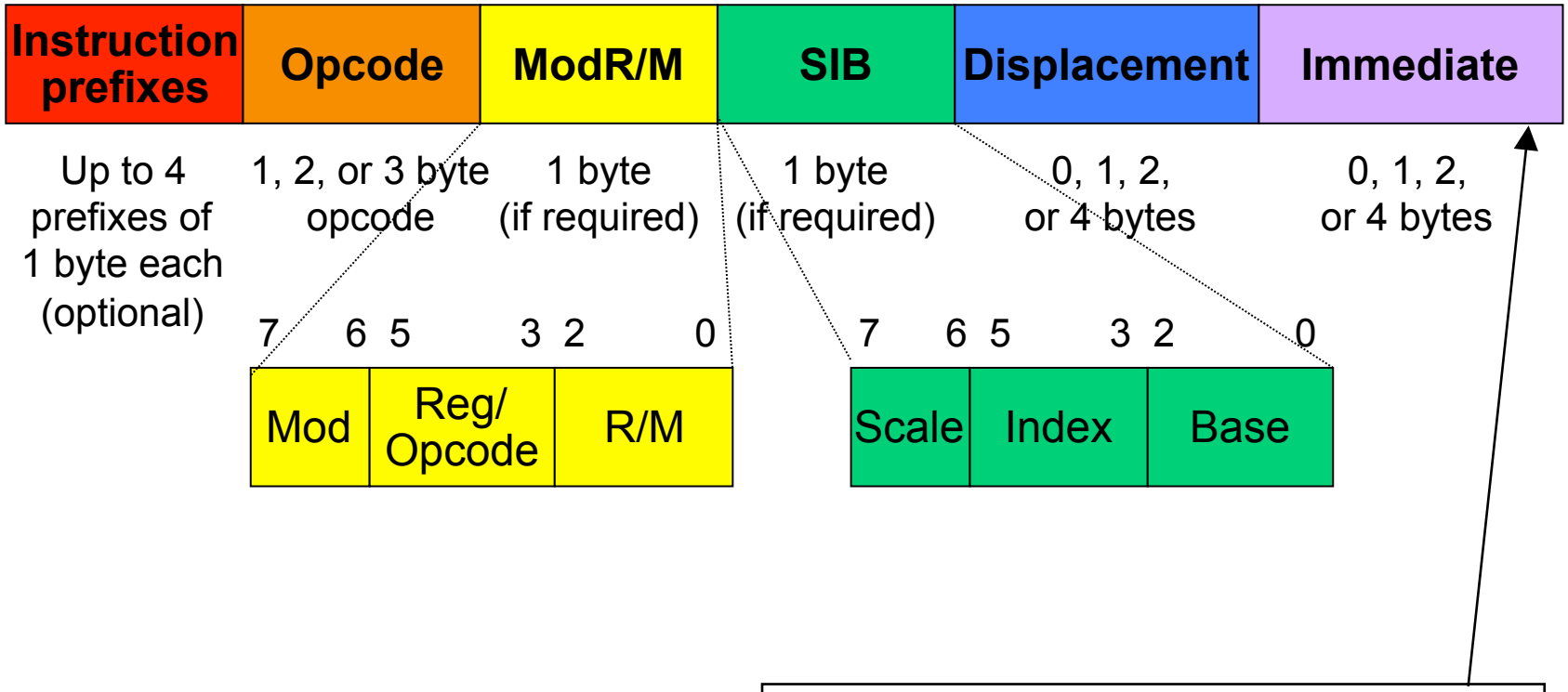
IA-32 Instruction Format (cont.)



Displacement

- Used in jump and call instructions
- Indicates the displacement between the destination instruction and the jump/call instruction
- More precisely, indicates:
  [addr of destination instr] – [addr of instr following the jump/call]
- Uses little-endian byte order

IA-32 Instruction Format (cont.)



Immediate

- Specifies an immediate operand
- Uses little-endian byte order

Example: Push on to Stack

- Assembly language:

```
pushl %edx
```

- Machine code:
  - IA32 has a separate opcode for push for each register operand
    - 50: pushl %eax
    - 51: pushl %ecx
    - 52: pushl %edx → 0101 0010
    - …
  - Results in a *one-byte* instruction
- Observe: sometimes one assembly language instruction can map to a *group* of different opcodes

Example: Load Effective Address

- Assembly language:

```
leal (%eax,%eax,4), %eax
```

- Machine code:
  - Byte 1: 8D (opcode for "load effective address")
  - Byte 2: 04 (dest %eax, with scale-index-base)
  - Byte 3: 80 (scale=4, index=%eax, base=%eax)

1000 1101

0000 0100

1000 0000

Load the address %eax + 4 * %eax into register %eax

CISC and RISC

- IA-32 machine language instructions are **complex**
- IA-32 is a
  - **Complex Instruction Set Computer (CISC)**
- Alternative:
  - **Reduced Instruction Set Computer (RISC)**

Characteristics of CISC and RISC

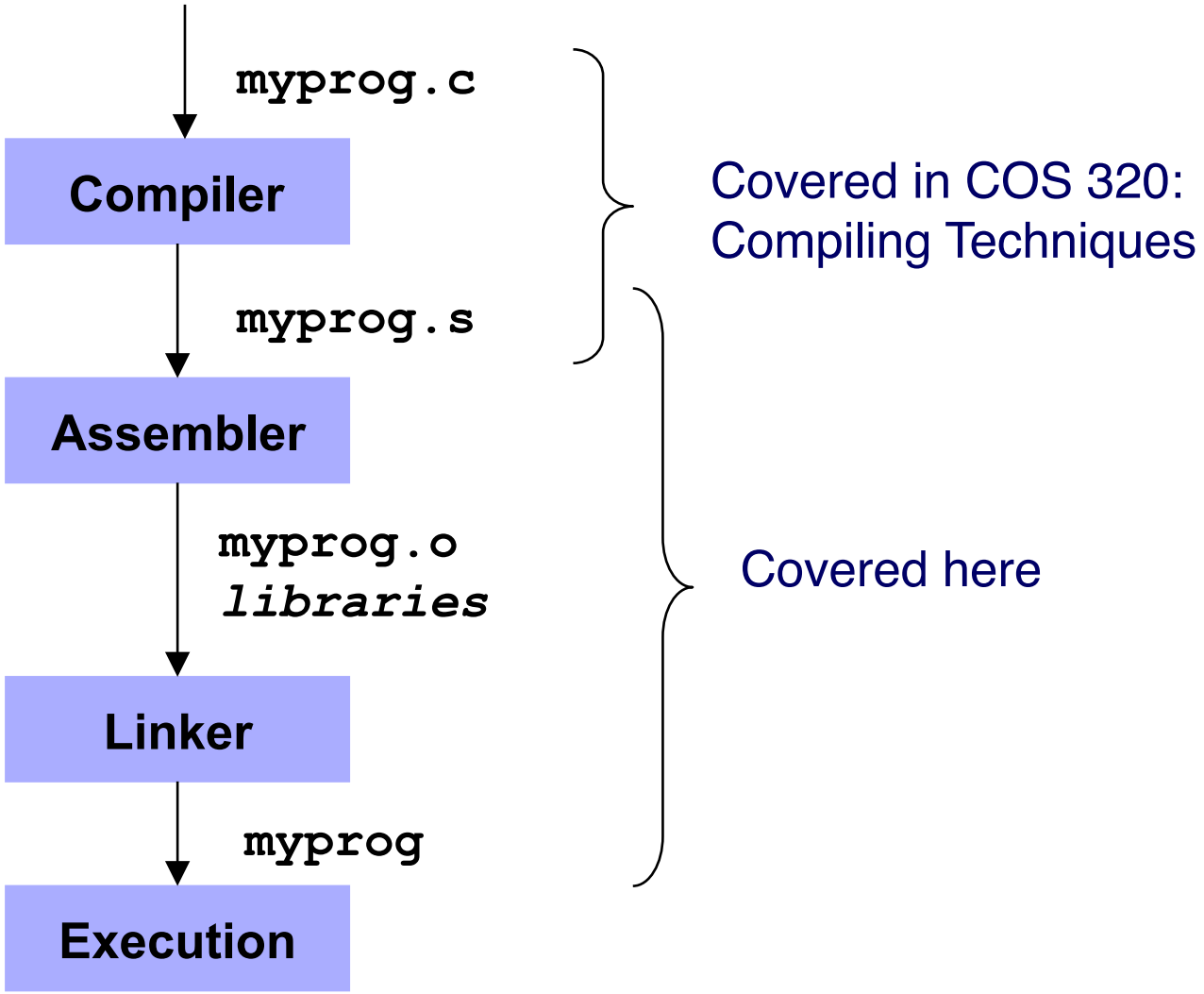
- CISC
  - **Many** instructions
  - **Many** addressing modes (direct, indirect, indexed, base-pointer)
  - Hardware interpretation is **complex**
  - **Few** instructions required to accomplish a given job (expressive)
  - Example: IA-32
- RISC
  - **Few** instructions
  - **Few** addressing modes (typically only direct and indirect)
  - Hardware interpretation is **simple**
  - **Many** instructions required for a given job (not expressive)
  - Examples: MIPS, SPARC

Brief History of CISC and RISC

- Stage 1: Programmers write assembly language
  - Important that assembly/machine language be expressive
  - CISC dominates (especially Intel)
- Stage 2: Programmers write high-level language
  - Not important that assembly/machine language be expressive; the compiler generates it
  - Important that compilers work well => assembly/machine language should be simple
  - RISC takes a foothold (but CISC, especially Intel, persists)
- Stage 3: Compilers get smarter
  - Less important that assembly/machine language be simple
  - Much motivation for RISC disappears
  - CISC (especially Intel) dominates the computing world

Part 2: The Assembly Process

The Build/Execute Process



Two Aspects of the Assembler/Linker

- Translating each instruction
  - Mapping an assembly-language instruction
  - … into the corresponding machine-language instruction
- Dealing with references across instructions
  - Jumps to other locations in same chunk of code
  - Accesses a global variable by the name of its memory location
  - Calling to and returning from functions defined in other code

```
main:
        pushl   %ebp
        movl    %esp, %ebp
        call    getchar
        cmpl    $'A', %eax
        jne     skip
        pushl   $msg
        call    printf
        addl    $4, %esp
skip:
        movl    $0, %eax
        movl    %ebp, %esp
        popl    %ebp
        ret
```

References Across Instructions

- Many instructions can be assembled independently
  - pushl %edx
  - leal (%eax, %eax, 4), %eax
  - movl $0, %eax
  - addl %ebx, %ecx
- But, some make references to other data or code
  - jne skip
  - pushl $msg
  - call printf
- Need to fill in those references
  - To generate a final executable binary

The Forward Reference Problem

- Problem

```
        …
        jmp mylabel
        …
mylabel:
        …
```

Any assembler must deal with the forward reference problem

- Assembler must generate machine language code for “jmp mylabel”
- But assembler hasn’t yet *seen* the definition of mylabel
  - I.e., the jmp instruction contains a forward reference to mylabel

The Forward Reference Solution

- Solution
  - Assembler performs **2 passes** over assembly language program
- Different assemblers perform different tasks in each pass
- One straightforward design…

Assembler Passes

- Pass1
  - Assembler traverses assembly program to create…
  - Symbol table
    - Key: label
    - Value: information about label
      - Label name, which section, what offset within that section, …
- Pass 2
  - Assembler traverses assembly program again to create…
  - RODATA section
  - DATA section
  - BSS section
  - TEXT section
  - Relocation record section
    - Each relocation record indicates an area the linker must patch

An Example Program

- A simple (nonsensical) program:

```
#include <stdio.h>
int main(void) {
   if (getchar() == 'A')
      printf("Hi\n");
   return 0;
}
```

- Let's consider how the assembler handles that program…

```
        .section ".rodata"
msg:
        .asciz  "Hi\n"
        .section ".text"
        .globl  main
main:
        pushl   %ebp
        movl    %esp, %ebp
        call    getchar
        cmpl    $'A', %eax
        jne     skip
        pushl   $msg
        call    printf
        addl    $4, %esp
skip:
        movl    $0, %eax
        movl    %ebp, %esp
        popl    %ebp
        ret
```

Assembler Data Structures (1)

- Symbol Table

Label	Section	Offset	Local?	Seq#

- Relocation Records

Section	Offset	Rel Type	Seq#

- RODATA Section (location counter: 0)

Offset	Contents	Explanation

- No DATA or BSS section in this program
- Initially all sections are empty

- TEXT Section (location counter: 0)

Offset	Contents	Explanation

Assembler Pass 1

```
        .section ".rodata"
msg:
        .asciz  "Hi\n"
        .section ".text"
        .globl  main
main:
        pushl   %ebp
        movl    %esp, %ebp
        call    getchar
        cmpl    $'A', %eax
        jne     skip
        pushl   $msg
        call    printf
        addl    $4, %esp
skip:
        movl    $0, %eax
        movl    %ebp, %esp
        popl    %ebp
        ret
```

Assembler notes that the current section is RODATA

Assembler adds binding to Symbol Table…

Assembler Data Structures (2)

- Symbol Table

Label	Section	Offset	Local?	Seq#
msg	RODATA	0	local	0

- msg marks a spot in the RODATA section at offset 0
- msg is a local label
- Assign msg sequence number 0

- Relocation Records
  - (Same)
- RODATA Section (location counter: 0)
  - (Same)
- TEXT Section (location counter: 0)
  - (Same)

Assembler Pass 1 (cont.)

```
        .section ".rodata"
msg:
        .asciz  "Hi\n"
        .section ".text"
        .globl  main
main:
        pushl   %ebp
        movl    %esp, %ebp
        call    getchar
        cmpl    $'A', %eax
        jne     skip
        pushl   $msg
        call    printf
        addl    $4, %esp
skip:
        movl    $0, %eax
        movl    %ebp, %esp
        popl    %ebp
        ret
```

Assembler increments RODATA section ***location counter*** by byte count of the string (4)…

Assembler Data Structures (3)

- Symbol Table

Label	Section	Offset	Local?	Seq#
msg	RODATA	0	local	0

- Relocation Records
  - (Same)
- RODATA Section (location counter: 4)
  - (Same)
- TEXT Section (location counter: 0)
  - (Same)

- RODATA location counter now is 4
- If another label were defined in at this point, it would mark a spot in RODATA at offset 4

Assembler Pass 1 (cont.)

```
        .section ".rodata"
msg:
        .asciz  "Hi\n"
        .section ".text"
        .globl  main
main:
        pushl   %ebp
        movl    %esp, %ebp
        call    getchar
        cmpl    $'A', %eax
        jne     skip
        pushl   $msg
        call    printf
        addl    $4, %esp
skip:
        movl    $0, %eax
        movl    %ebp, %esp
        popl    %ebp
        ret
```

Assembler notes that current section is TEXT

Assembler does nothing

Assembler adds binding to Symbol Table…

Assembler Data Structures (4)

- Symbol Table

Label	Section	Offset	Local?	Seq#
msg	RODATA	0	local	0
main	TEXT	0	local	1

- main marks a spot in the TEXT section at offset 0
- main is a local label (assembler will discover otherwise in Pass 2)
- Assign main sequence number 1

- Relocation Records
  - (Same)
- RODATA Section (location counter: 4)
  - (Same)
- TEXT Section (location counter: 0)
  - (Same)

Assembler Pass 1 (cont.)

```
        .section ".rodata"
msg:
        .asciz  "Hi\n"
        .section ".text"
        .globl  main
main:
        pushl   %ebp
        movl    %esp, %ebp
        call    getchar
        cmpl    $'A', %eax
        jne     skip
        pushl   $msg
        call    printf
        addl    $4, %esp
skip:
        movl    $0, %eax
        movl    %ebp, %esp
        popl    %ebp
        ret
```

Assembler increments TEXT section location counter by the length of each instruction…

Assembler Data Structures (5)

- Symbol Table

Label	Section	Offset	Local?	Seq#
msg	RODATA	0	local	0
main	TEXT	0	local	1

- Relocation Records
  - (Same)
- RODATA Section (location counter: 4)
  - (Same)
- TEXT Section (location counter: 26)
  - (Same)

- TEXT location counter now is 26
- If another label were defined at this point, it would mark a spot in TEXT at offset 26

Assembler Pass 1 (cont.)

```
        .section ".rodata"
msg:
        .asciz  "Hi\n"
        .section ".text"
        .globl  main
main:
        pushl   %ebp
        movl    %esp, %ebp
        call    getchar
        cmpl    $'A', %eax
        jne     skip
        pushl   $msg
        call    printf
        addl    $4, %esp
skip:
        movl    $0, %eax
        movl    %ebp, %esp
        popl    %ebp
        ret
```

Assembler adds binding to Symbol Table…

Assembler Data Structures (6)

- Symbol Table

Label	Section	Offset	Local?	Seq#
msg	RODATA	0	local	0
main	TEXT	0	local	1
skip	TEXT	26	local	2

- skip marks a spot in the TEXT section at offset 26
- skip is a local label
- Assign skip sequence number 2

- Relocation Records
  - (Same)
- RODATA Section (location counter: 4)
  - (Same)
- TEXT Section (location counter: 26)
  - (Same)

Assembler Pass 1 (cont.)

```
        .section ".rodata"
msg:
        .asciz  "Hi\n"
        .section ".text"
        .globl  main
main:
        pushl   %ebp
        movl    %esp, %ebp
        call    getchar
        cmpl    $'A', %eax
        jne     skip
        pushl   $msg
        call    printf
        addl    $4, %esp
skip:
        movl    $0, %eax
        movl    %ebp, %esp
        popl    %ebp
        ret
```

Assembler increments TEXT section location counter by the length of each instruction…

Assembler Data Structures (7)

- Symbol Table

Label	Section	Offset	Local?	Seq#
msg	RODATA	0	local	0
main	TEXT	0	local	1
skip	TEXT	26	local	2

- Relocation Records
  - (Same)
- RODATA Section (location counter: 4)
  - (Same)
- TEXT Section (location counter: 35)
  - (Same)

- TEXT location counter now is 35
- If another label were defined at this point, it would mark a spot in TEXT at offset 35

From Assembler Pass 1 to Pass 2

- End of Pass 1
  - Assembler has (partially) created Symbol Table
  - So assembler now knows which location each label marks
- Beginning of Pass 2
  - Assembler resets all section location counters…

Assembler Data Structures (8)

- Symbol Table

Label	Section	Offset	Local?	Seq#
msg	RODATA	0	local	0
main	TEXT	0	local	1
skip	TEXT	26	local	2

- Relocation Records
  - (Same)
- RODATA Section (location counter: 0)
  - (Same)
- TEXT Section (location counter: 0)
  - (Same)

- Location counters reset to 0

Assembler Pass 2

```
        .section ".rodata"
msg:
        .asciz  "Hi\n"
        .section ".text"
        .globl  main
main:
        pushl   %ebp
        movl    %esp, %ebp
        call    getchar
        cmpl    $'A', %eax
        jne     skip
        pushl   $msg
        call    printf
        addl    $4, %esp
skip:
        movl    $0, %eax
        movl    %ebp, %esp
        popl    %ebp
        ret
```

Assembler notes that the current section is RODATA

Assembler does nothing

Assembler places bytes in RODATA section, and increments location counter…

Assembler Data Structures (9)

- Symbol Table
  - (Same)
- Relocation Records
  - (Same)
- RODATA Section (location counter: 4)

Offset	Contents (hex)	Explanation
0	48	ASCII code for 'H'
1	69	ASCII code for 'i'
2	0A	ASCII code for '\n'
3	00	ASCII code for null char

- TEXT Section (location counter: 0)
  - (Same)

- Location counter incremented to 4

- RODATA section contains the bytes comprising the string

Assembler Pass 2 (cont.)

```
        .section ".rodata"
msg:
        .asciz  "Hi\n"
        .section ".text"
        .globl  main
main:
        pushl   %ebp
        movl    %esp, %ebp
        call    getchar
        cmpl    $'A', %eax
        jne     skip
        pushl   $msg
        call    printf
        addl    $4, %esp
skip:
        movl    $0, %eax
        movl    %ebp, %esp
        popl    %ebp
        ret
```

Assembler notes that the current section is TEXT

Assembler updates Symbol Table…

Assembler Data Structures (10)

- Symbol Table

Label	Section	Offset	Local?	Seq#
msg	RODATA	0	local	0
main	TEXT	0	global	1
skip	TEXT	26	local	2

- main is a global label

- Relocation Records
  - (Same)
- RODATA Section (location counter: 4)
  - (Same)
- TEXT Section (location counter: 0)
  - (Same)

Assembler Pass 2 (cont.)

```
        .section ".rodata"
msg:
        .asciz  "Hi\n"
        .section ".text"
        .globl  main
main:
        pushl   %ebp
        movl    %esp, %ebp
        call    getchar
        cmpl    $'A', %eax
        jne     skip
        pushl   $msg
        call    printf
        addl    $4, %esp
skip:
        movl    $0, %eax
        movl    %ebp, %esp
        popl    %ebp
        ret
```

Assembler does nothing

Assembler generates machine language code in current (TEXT) section…

Assembler Data Structures (11)

- Symbol Table
  - (Same)
- Relocation Records
  - (Same)
- RODATA Section (location counter: 4)
  - (Same)
- TEXT Section (location counter: 1)

Offset	Contents	Explanation
0	55	pushl %ebp 01010101 This is a "pushl %ebp" instruction

Assembler Pass 2 (cont.)

```
        .section ".rodata"
msg:
        .asciz  "Hi\n"
        .section ".text"
        .globl  main
main:
        pushl   %ebp
        movl    %esp, %ebp
        call    getchar
        cmpl    $'A', %eax
        jne     skip
        pushl   $msg
        call    printf
        addl    $4, %esp
skip:
        movl    $0, %eax
        movl    %ebp, %esp
        popl    %ebp
        ret
```

Assembler generates machine language code in current (TEXT) section…

Assembler Data Structures (12)

- Symbol Table
  - (Same)
- Relocation Records
  - (Same)
- RODATA Section (location counter: 4)
  - (Same)
- TEXT Section (location counter: 3)

Offset	Contents	Explanation
…	…	…
1-2	89 E5	movl %esp,%ebp 10001001 11 100 101 This is a "movl" instruction whose source operand is a register The M field designates a register The source register is ESP The destination register is EBP

Assembler Pass 2 (cont.)

```
        .section ".rodata"
msg:
        .asciz  "Hi\n"
        .section ".text"
        .globl  main
main:
        pushl   %ebp
        movl    %esp, %ebp
        call    getchar
        cmpl    $'A', %eax
        jne     skip
        pushl   $msg
        call    printf
        addl    $4, %esp
skip:
        movl    $0, %eax
        movl    %ebp, %esp
        popl    %ebp
        ret
```

Assembler generates machine language code in current (TEXT) section…

Assembler Data Structures (12)

- Symbol Table
  - (Same)
- Relocation Records
  - (Same)
- RODATA Section (location counter: 4)
  - (Same)
- TEXT Section (location counter: 8)

- Assembler looks in Symbol Table to find offset of getchar
- getchar is not in Symbol Table
- Assembler cannot compute displacement that belongs at offset 4
- So…

Offset	Contents	Explanation
…	…	…
3-7	E8 ????????	call getchar 11101000 ???????????????????????????????? This is a "call" instruction with a 4-byte immmediate operand This the displacement

Assembler Data Structures (13)

- Symbol Table

Label	Section	Offset	Local?	Seq#
msg	RODATA	0	local	0
main	TEXT	0	global	1
skip	TEXT	26	local	2
getchar	?	?	global	3

- Relocation Records
  - (Same)
- RODATA Section (location counter: 4)
  - (Same)
- TEXT Section (location counter: 8)
  - (Same)

- Assembler adds getchar to Symbol Table
- Then…

Assembler Data Structures (14)

- Symbol Table
  - (Same)
- Relocation Records

Section	Offset	Rel Type	Seq#
TEXT	4	displacement	3

- Assembler generates a relocation record, thus asking linker to patch code

Dear Linker,
Please patch the TEXT section at offset 4. Do a "displacement" type of patch. The patch is with respect to the label whose seq number is 3 (i.e. getchar).
Sincerely,
Assembler

- RODATA Section (location counter: 4)
  - (Same)
- TEXT Section (location counter: 8)
  - (Same)

Assembler Pass 2 (cont.)

```
        .section ".rodata"
msg:
        .asciz  "Hi\n"
        .section ".text"
        .globl  main
main:
        pushl   %ebp
        movl    %esp, %ebp
        call    getchar
        cmpl    $'A', %eax
        jne     skip
        pushl   $msg
        call    printf
        addl    $4, %esp
skip:
        movl    $0, %eax
        movl    %ebp, %esp
        popl    %ebp
        ret
```

Assembler generates machine language code in current (TEXT) section…

Assembler Data Structures (15)

- Symbol Table
  - (Same)
- Relocation Records
  - (Same)
- RODATA Section (location counter: 4)
  - (Same)
- TEXT Section (location counter: 11)

Offset	Contents	Explanation
…	…	…
8-10	83 F8 41	cmpl %'A',%eax 10000011 11 111 000 01000001 This is some "1" instruction that has a 1 byte immediate operand The M field designates a register This is a "cmp" instruction The destination register is EAX The immediate operand is 'A'

Assembler Pass 2 (cont.)

```
        .section ".rodata"
msg:
        .asciz  "Hi\n"
        .section ".text"
        .globl  main
main:
        pushl   %ebp
        movl    %esp, %ebp
        call    getchar
        cmpl    $'A', %eax
        jne     skip
        pushl   $msg
        call    printf
        addl    $4, %esp
skip:
        movl    $0, %eax
        movl    %ebp, %esp
        popl    %ebp
        ret
```

Assembler generates machine language code in current (TEXT) section…

Assembler Data Structures (16)

- Symbol Table
  - (Same)
- Relocation Records
  - (Same)
- RODATA Section (location counter: 4)
  - (Same)
- TEXT Section (location counter: 13)

- Assembler looks in Symbol Table to find offset of skip (26)
- Assembler subtracts offset of next instruction (13)
- Resulting displacement is 13

Offset	Contents	Explanation
…	…	…
11-12	75 0D	jne skip 01110101 00001101 This is a jne instruction that has a 1 byte immediate operand The displacement between the destination instr. and the next instr. is 13

Assembler Pass 2 (cont.)

```
        .section ".rodata"
msg:
        .asciz  "Hi\n"
        .section ".text"
        .globl  main
main:
        pushl   %ebp
        movl    %esp, %ebp
        call    getchar
        cmpl    $'A', %eax
        jne     skip
        pushl   $msg
        call    printf
        addl    $4, %esp
skip:
        movl    $0, %eax
        movl    %ebp, %esp
        popl    %ebp
        ret
```

Assembler generates machine language code in current (TEXT) section…

Assembler Data Structures (16)

- Symbol Table
  - (Same)
- Relocation Records
  - (Same)
- RODATA Section (location counter: 4)
  - (Same)
- TEXT Section (location counter: 18)

- Assembler knows offset of msg (0) within RODATA section
- But assembler does not know location of RODATA section
- So assembler does not know location of msg
- So…

Offset	Contents	Explanation
…	…	…
13-17	68 ????????	pushl $msg 001101000 ???????????????????????????????? This is a pushl instruction with a 4 byte immediate operand This is the data to be pushed

Assembler Data Structures (17)

- Symbol Table
  - (Same)
- Relocation Records

Section	Offset	Rel Type	Seq#
…	…	…	…
TEXT	14	absolute	0

- Assembler generates a relocation record, thus asking linker to patch code

- RODATA Section (location counter: 4)
  - (Same)
- TEXT Section (location counter: 18)
  - (Same)

Dear Linker,
Please patch the TEXT section at offset 14. Do an "absolute" type of patch. The patch is with respect to the label whose seq number is 0 (i.e. msg).
Sincerely,
Assembler

Assembler Pass 2 (cont.)

```
        .section ".rodata"
msg:
        .asciz  "Hi\n"
        .section ".text"
        .globl  main
main:
        pushl   %ebp
        movl    %esp, %ebp
        call    getchar
        cmpl    $'A', %eax
        jne     skip
        pushl   $msg
        call    printf
        addl    $4, %esp
skip:
        movl    $0, %eax
        movl    %ebp, %esp
        popl    %ebp
        ret
```

Assembler generates machine language code in current (TEXT) section…

Assembler Data Structures (18)

- Symbol Table
  - (Same)
- Relocation Records
  - (Same)
- RODATA Section (location counter: 4)
  - (Same)
- TEXT Section (location counter: 23)

- Assembler looks in Symbol Table to find offset of printf
- printf is not in Symbol Table
- Assembler cannot compute displacement that belongs at offset 19
- So…

Offset	Contents	Explanation
…	…	…
18-22	E8 ????????	call printf 11101000 ???????????????????????????? This is a "call" instruction with a 4-byte immmediate operand This the displacement

Assembler Data Structures (19)

- Symbol Table

Label	Section	Offset	Local?	Seq#
msg	RODATA	0	local	0
main	TEXT	0	global	1
skip	TEXT	26	local	2
getchar	?	?	global	3
printf	?	?	global	4

- Relocation Records
  - (Same)
- RODATA Section (location counter: 4)
  - (Same)
- TEXT Section (location counter: 23)
  - (Same)

- Assembler adds printf to Symbol Table
- Then…

Assembler Data Structures (20)

- Symbol Table
  - (Same)
- Relocation Records

Section	Offset	Rel Type	Seq#
…	…	…	…
TEXT	19	displacement	4

- Assembler generates a relocation record, thus asking linker to patch code

Dear Linker,
Please patch the TEXT section at offset 19. Do a "displacement" type of patch. The patch is with respect to the label whose seq number is 4 (i.e. printf).
Sincerely,
Assembler

- RODATA Section (location counter: 4)
  - (Same)
- TEXT Section (location counter: 8)
  - (Same)

Assembler Pass 2 (cont.)

```
        .section ".rodata"
msg:
        .asciz  "Hi\n"
        .section ".text"
        .globl  main
main:
        pushl   %ebp
        movl    %esp, %ebp
        call    getchar
        cmpl    $'A', %eax
        jne     skip
        pushl   $msg
        call    printf
        addl    $4, %esp
skip:
        movl    $0, %eax
        movl    %ebp, %esp
        popl    %ebp
        ret
```

Assembler ignores

Assembler generates machine language code in current (TEXT) section…

Assembler Data Structures (21)

- Symbol Table, Relocation Records, RODATA Section
  - (Same)
- TEXT Section (location counter: 31)

Offset	Contents	Explanation
…	…	…
23-25	83 C4 04	addl $4,%esp 10000011 11 000 100 00000100 This is some "1" instruction that has a 1 byte immediate operand The M field designates a register This is an "add" instruction The destination register is ESP The immediate operand is 4
26-30	B8 00000000	movl $0,%eax 10111000 00000000000000000000000000000000 This is an instruction of the form "movl 4-byte-immediate, %eax" The immediate operand is 0

Assembler Data Structures (22)

- Symbol Table, Relocation Records, RODATA Section
  - (Same)
- TEXT Section (location counter: 35)

Offset	Contents	Explanation
…	…	…
31-32	89 EC	movl %ebp,%esp 10001001 11 101 100 This is a "movl" instruction whose source operand is a register The M field designates a register The source register is EBP The destination register is ESP
33	5D	popl %ebp 01011101 This is a "popl %ebp" instruction
34	C3	ret 11000011 This is a "ret" instruction

From Assembler to Linker

- Assembler writes its data structures to .o file
- Linker:
  - Reads .o file
  - Works in two phases: resolution and relocation

Linker Resolution

- Resolution
  - Linker resolves references
- For this program, linker:
  - Notes that Symbol Table contains undefined labels
    - getchar and printf
  - Fetches, from libc.a, machine language code defining getchar and printf
  - Adds that code to TEXT section
    - (May add code to other sections too)
  - Updates Symbol Table to note offsets of getchar and printf
  - Adds column to Symbol Table to note addresses of all labels

Linker Relocation

- Relocation
  - Linker patches ("relocates") code
  - Linker traverses relocation records, patching code as specified
- For this program

Section	Offset	Rel Type	Seq#
TEXT	4	displacement	3
TEXT	14	absolute	0
TEXT	19	displacement	4

- Linker looks up offset of getchar
- Linker computes:
  [offset of getchar] – 8
- Linker places difference in TEXT section at offset 4

Linker Relocation (cont.)

- For this program

Section	Offset	Rel Type	Seq#
TEXT	4	displacement	3
TEXT	14	absolute	0
TEXT	19	displacement	4

- Linker looks up addr of msg
- Linker places addr in TEXT section at offset 14

Linker Relocation (cont.)

- For this program

Section	Offset	Rel Type	Seq#
TEXT	4	displacement	3
TEXT	14	absolute	0
TEXT	19	displacement	4

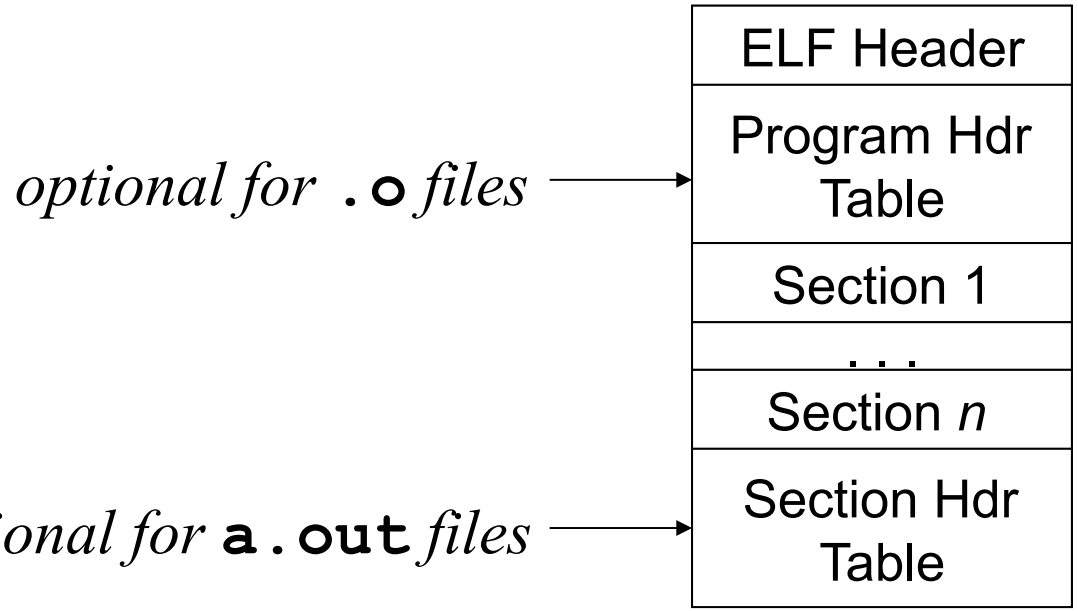
- Linker looks up offset of printf
- Linker computes:
  [offset of printf] – 23
- Linker places difference in TEXT section at offset 19

Linker Finishes

- Linker writes resulting TEXT, RODATA, DATA, BSS sections to executable binary file

ELF: Executable and Linking Format

- Unix format of object and executable files
  - Output by the assembler
  - Input and output of linker



Conclusions

- Assembler: reads assembly language file
  - Pass 1: Generates Symbol Table
    - Contains info about labels
  - Pass 2: Uses Symbol Table to generate code
    - TEXT, RODATA, DATA, BSS sections
    - Relocation Records
  - Writes object file (ELF)
- Linker: reads object files
  - Resolution: Resolves references to make Symbol Table complete
  - Relocation: Uses Symbol Table and Relocation Records to patch code
  - Writes executable binary file (ELF)

Appendix: Generating Machine Lang

- Hint for Buffer Overrun assignment…
- Given an assembly language instruction, how can you find the machine language equivalent?
- Option 1: Consult IA-32 reference manuals
  - See course Web pages for links to the manuals

Appendix: Generating Machine Lang

- Option 2:
  - Compose an assembly language program that contains the given assembly language instruction
  - Then use gdb…

Appendix: Generating Machine Lang

- Using gdb

```
$ gcc217 detecta.s -o detecta
$ gdb detecta
(gdb) x/12i main
0x80483b4 <main>:       push   %ebp
0x80483b5 <main+1>:     mov    %esp,%ebp
0x80483b7 <main+3>:     call   0x8048298 <getchar@plt>
0x80483bc <main+8>:     cmp    $0x41,%eax
0x80483bf <main+11>:    jne    0x80483ce <skip>
0x80483c1 <main+13>:    push   $0x80484b0
0x80483c6 <main+18>:    call   0x80482c8 <printf@plt>
0x80483cb <main+23>:    add    $0x4,%esp
0x80483ce <skip>:       mov    $0x0,%eax
0x80483d3 <skip+5>:     mov    %ebp,%esp
0x80483d5 <skip+7>:     pop    %ebp
0x80483d6 <skip+8>:     ret
(gdb) x/35b main
0x0 <main>:      0x55   0x89   0xe5   0xe8   0xfc   0xff   0xff   0xff
0x8 <main+8>:    0x83   0xf8   0x41   0x75   0x0d   0x68   0x00   0x00
0x10 <main+16>:  0x00   0x00   0xe8   0xfc   0xff   0xff   0xff   0x83
0x18 <main+24>:  0xc4   0x04   0xb8   0x00   0x00   0x00   0x00   0x89
0x20 <skip+6>:   0xec   0x5d   0xc3
(gdb) quit
```

Build program; run gdb from shell

Issue x/i command to examine memory as instructions

Issue x/b command to examine memory as raw bytes

Match instructions to bytes

Appendix: Generating Machine Lang

- Option 3:
  - Compose an assembly language program that contains the given assembly language instruction
  - Then use objdump – a special purpose tool…

Appendix: Generating Machine Lang

- Using objdump

Build program; run objdump

Machine language

Assembly language

```
$ gcc217 detecta.s -o detecta
$ objdump -d detecta
detecta:     file format elf32-i386
…
Disassembly of section .text:
…
080483b4 <main>:
 80483b4:       55                      push   %ebp
 80483b5:       89 e5                   mov    %esp,%ebp
 80483b7:       e8 dc fe ff ff          call   8048298 <getchar@plt>
 80483bc:       83 f8 41                cmp    $0x41,%eax
 80483bf:       75 0d                   jne    80483ce <skip>
 80483c1:       68 b0 84 04 08          push   $0x80484b0
 80483c6:       e8 fd fe ff ff          call   80482c8 <printf@plt>
 80483cb:       83 c4 04                add    $0x4,%esp

080483ce <skip>:
 80483ce:       b8 00 00 00 00          mov    $0x0,%eax
 80483d3:       89 ec                   mov    %ebp,%esp
 80483d5:       5d                      pop    %ebp
 80483d6:       c3                      ret
…
```

Exceptions and Processes

Jennifer Rexford

The material for this lecture is drawn from
Computer Systems: A Programmer's Perspective (Bryant & O'Hallaron) Chapter 8

Goals of this Lecture

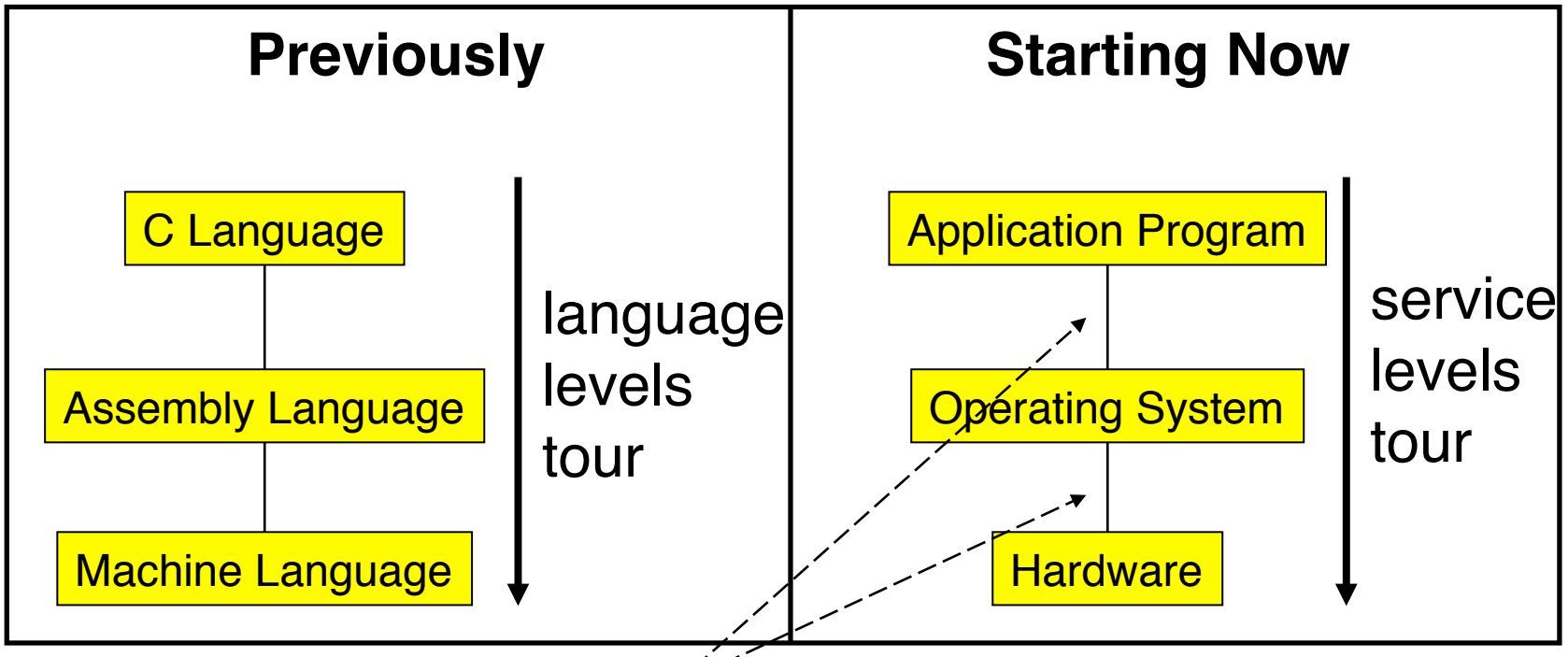
- Help you learn about:
  - **Exceptions**
  - The **process** concept

  … and thereby…
  - How operating systems work
  - How applications interact with OS and hardware

The **process** concept is one of the most important concepts in systems programming

Context of this Lecture

Second half of the course



Application programs, OS, and hardware interact via **exceptions**

Motivation

Question:

- How does a program get input from the keyboard?
- How does a program get data from a (slow) disk?

Question:

- Executing program thinks it has exclusive control of CPU
- But multiple programs share one CPU (or a few CPUs)
- How is that illusion implemented?

Question:

- Executing program thinks it has exclusive use of memory
- But multiple programs must share one memory
- How is that illusion implemented?

Answers: Exceptions…

Exceptions

- **Exception**
  - An abrupt change in control flow in response to a change in processor state
- Examples:
  - Application program:
    - Requests I/O
    - Requests more heap memory
    - Attempts integer division by 0
    - Attempts to access privileged memory
    - Accesses variable that is not in real memory (see upcoming "Virtual Memory" lecture)
  - User presses key on keyboard
  - Disk controller finishes reading data

Synchronous

Asynchronous

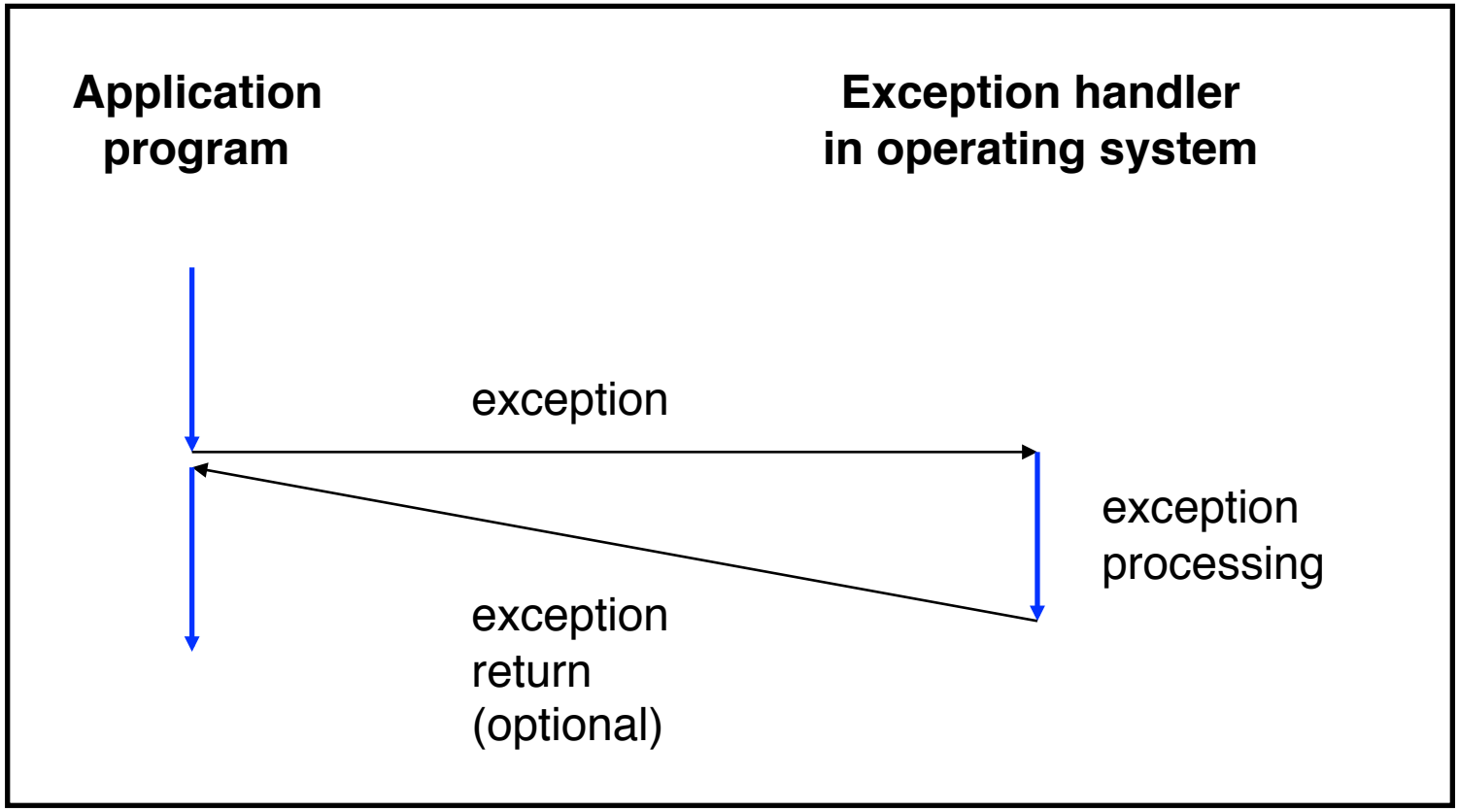
Exceptions Note

- Note:

  Exceptions in OS ≠ exceptions in Java

  Implemented using
  `try/catch`
  and **`throw`** statements

Exceptional Control Flow



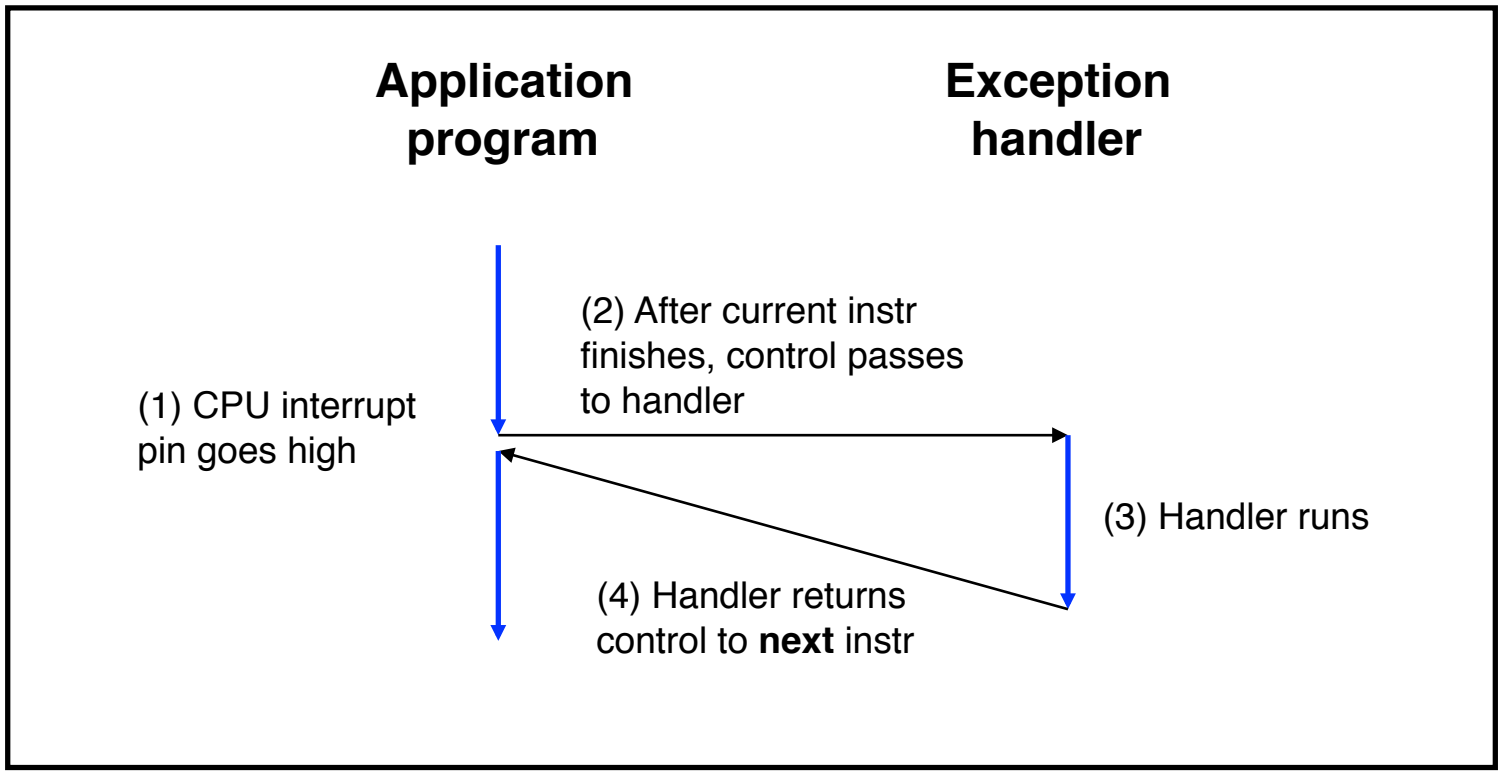
Exceptions vs. Function Calls

- Exceptions are **similar to** function calls
  - Control transfers from original code to other code
  - Other code executes
  - Control returns to original code
- Exceptions are **different from** function calls
  - Processor pushes **additional state** onto stack
    - E.g. values of *all* registers (including EFLAGS)
  - Processor pushes data onto **OS's stack**, not application's stack
  - Handler runs in **privileged mode**, not in **user mode**
    - Handler can execute all instructions and access all memory
  - Control **might return** to next instruction
    - Control sometimes returns to **current** instruction
    - Control sometimes does not return at all!

Classes of Exceptions

- There are four classes of exceptions…
  - Interrupts
  - Traps
  - Faults
  - Aborts

(1) Interrupts


Application program
Exception handler
(1) CPU interrupt pin goes high
(2) After current instr finishes, control passes to handler
(3) Handler runs
(4) Handler returns control to **next** instr

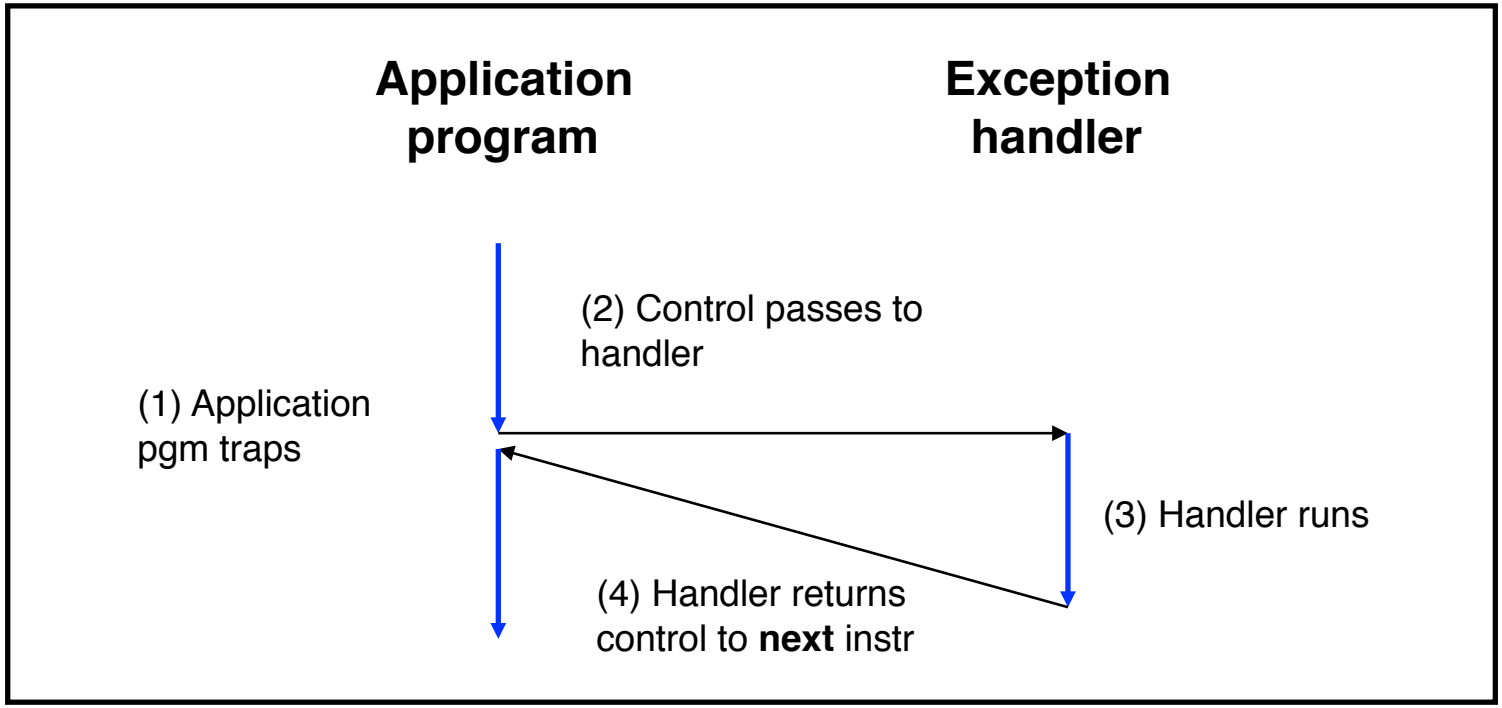
Cause: Signal from I/O device

Examples:

- User presses key
- Disk controller finishes reading/writing data
- Timer to trigger another application to run

An alternative to wasteful polling!

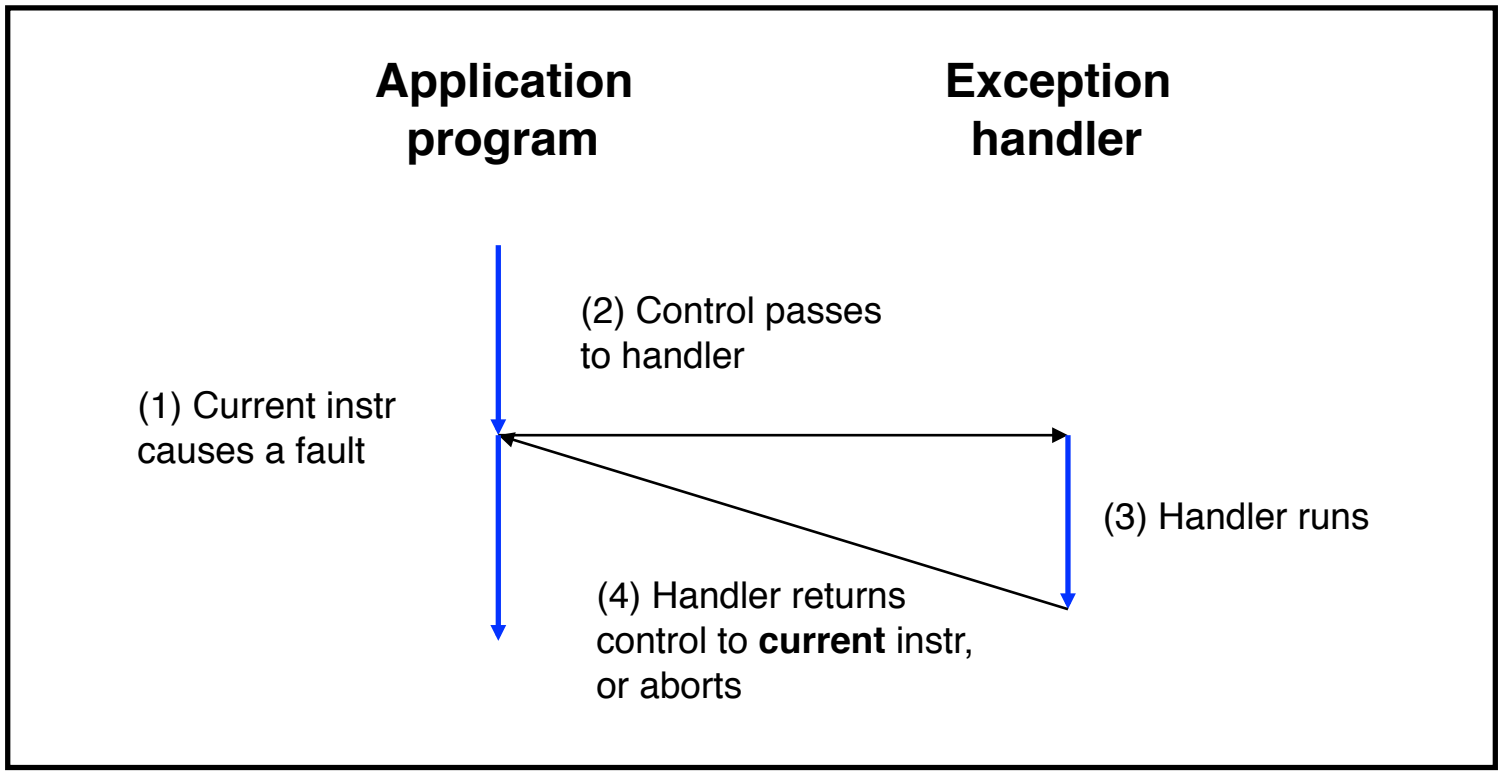
(2) Traps



Cause: Intentional (application program requests OS service)
Examples:
Application program requests more heap memory
Application program requests I/O

Traps provide a function-call-like interface between application and OS

(3) Faults


Application program
Exception handler
(1) Current instr causes a fault
(2) Control passes to handler
(3) Handler runs
(4) Handler returns control to **current** instr, or aborts

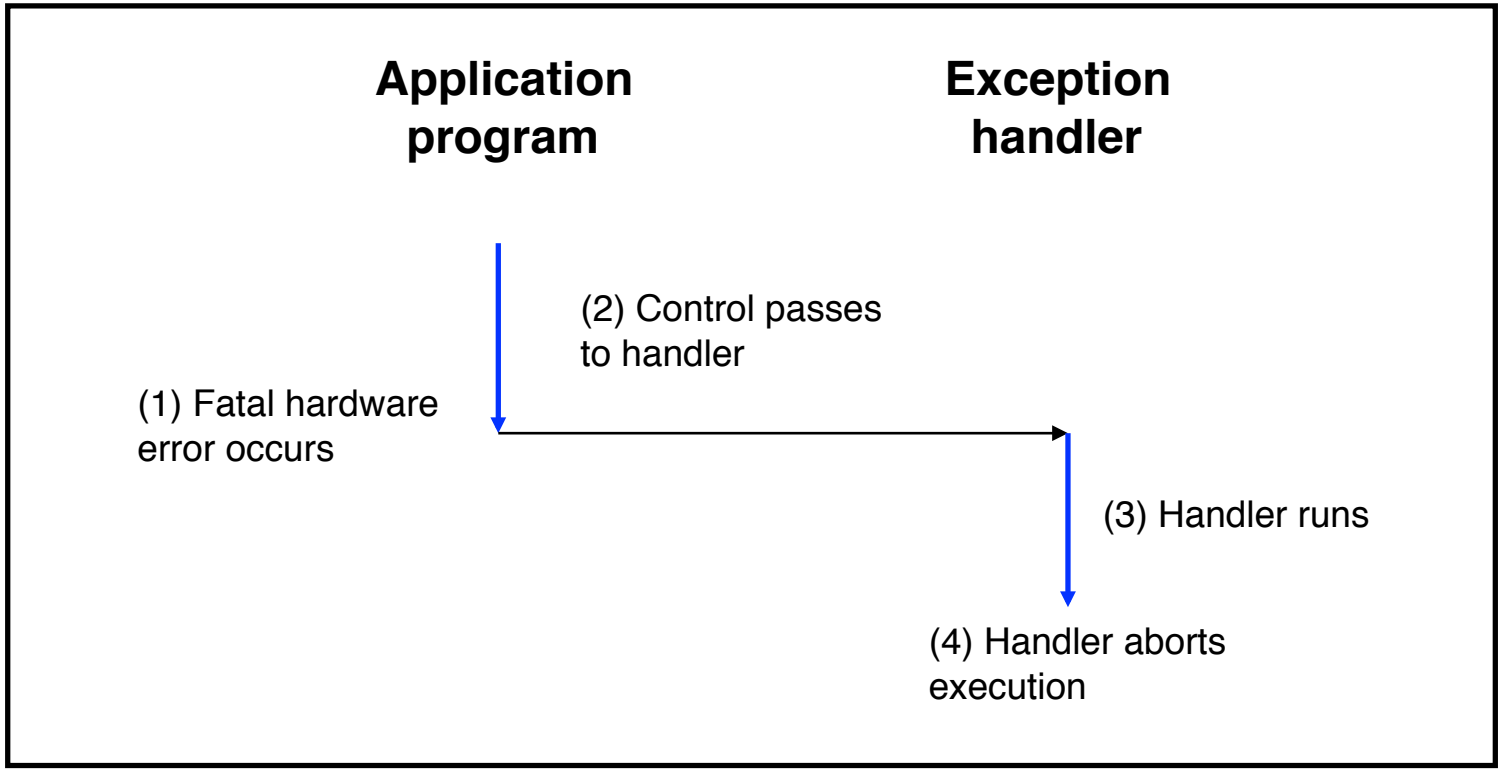
Cause: Application program causes (possibly) recoverable error

Examples:

Application program accesses privileged memory (segmentation fault)

Application program accesses data that is not in real memory (page fault)

(4) Aborts



Cause: Non-recoverable error

Example:

Parity check indicates corruption of memory bit (overheating, cosmic ray!, etc.)

Summary of Exception Classes

Class	Cause	Asynch/Synch	Return Behavior
Interrupt	Signal from I/O device	Asynch	Return to next instr
Trap	Intentional	Sync	Return to next instr
Fault	(Maybe) recoverable error	Sync	(Maybe) return to current instr
Abort	Non-recoverable error	Sync	Do not return

Exceptions in Intel Processors

Each exception has a number
Some exceptions in Intel processors:

Exception #	Exception
0	Fault: Divide error
13	Fault: Segmentation fault
14	Fault: Page fault (see “Virtual Memory” lecture)
18	Abort: Machine check
32-127	Interrupt or trap (OS-defined)
128	**Trap**
129-255	Interrupt or trap (OS-defined)

Traps in Intel Processors

- To execute a trap, application program should:
  - Place number in EAX register indicating desired functionality
  - Place parameters in EBX, ECX, EDX registers
  - Execute assembly language instruction “int 128”
- Example: To request more heap memory…

In Linux, 45 indicates request for more heap memory

```
movl    $45, %eax
movl    $1024, %ebx
int     $128
```

Causes trap

Request is for 1024 bytes

System-Level Functions

- For convenience, traps are wrapped in **system-level functions**
- Example: To request more heap memory…

```
/* unistd.h */
void *sbrk(intptr_t increment);
…
```

`sbrk()` is a system-level function

```
/* unistd.s */
Defines sbrk() in assembly lang
Executes int instruction
…
```

```
/* client.c */
…
sbrk(1024);
…
```

A call of a system-level function, that is, a **system call**

See Appendix for list of some Linux system-level functions

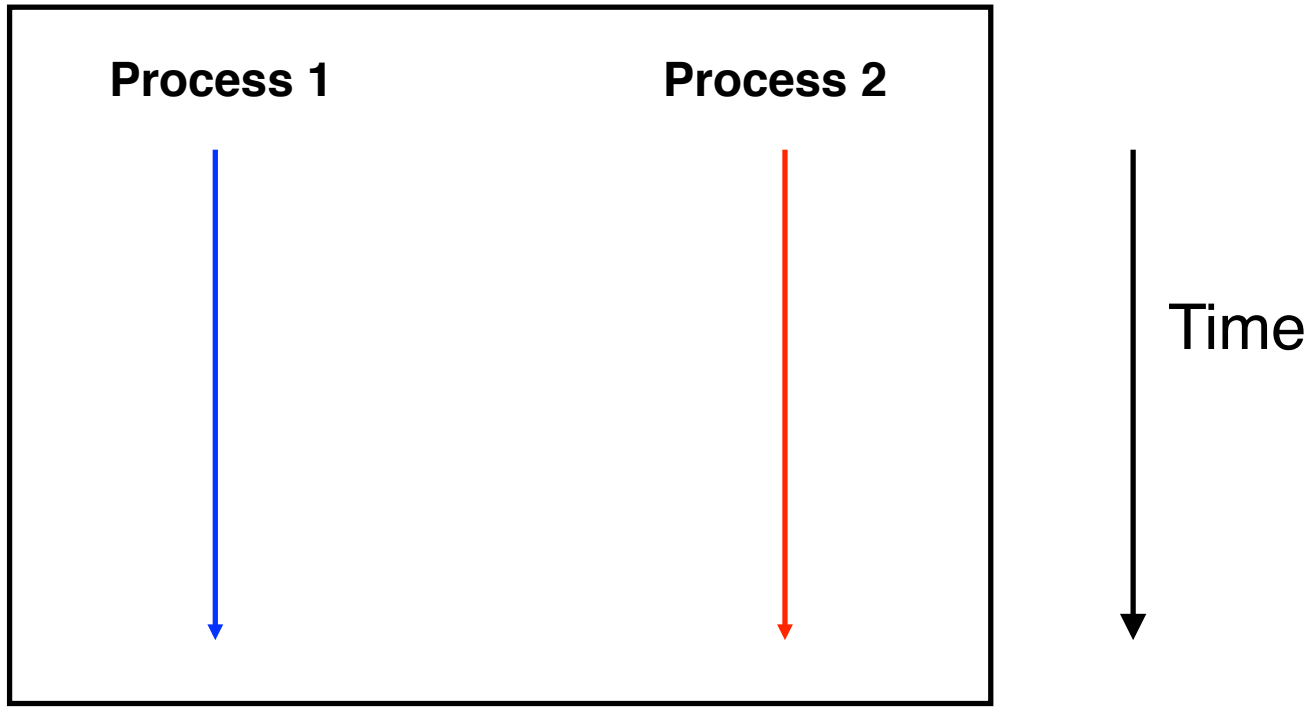
Processes

- **Program**
  - Executable code
- **Process**
  - An instance of a program in execution
- Each program runs in the **context** of some process
- **Context** consists of:
  - Process ID
  - Address space
    - TEXT, RODATA, DATA, BSS, HEAP, and STACK
  - Processor state
    - EIP, EFLAGS, EAX, EBX, etc. registers
  - Etc.

Significance of Processes

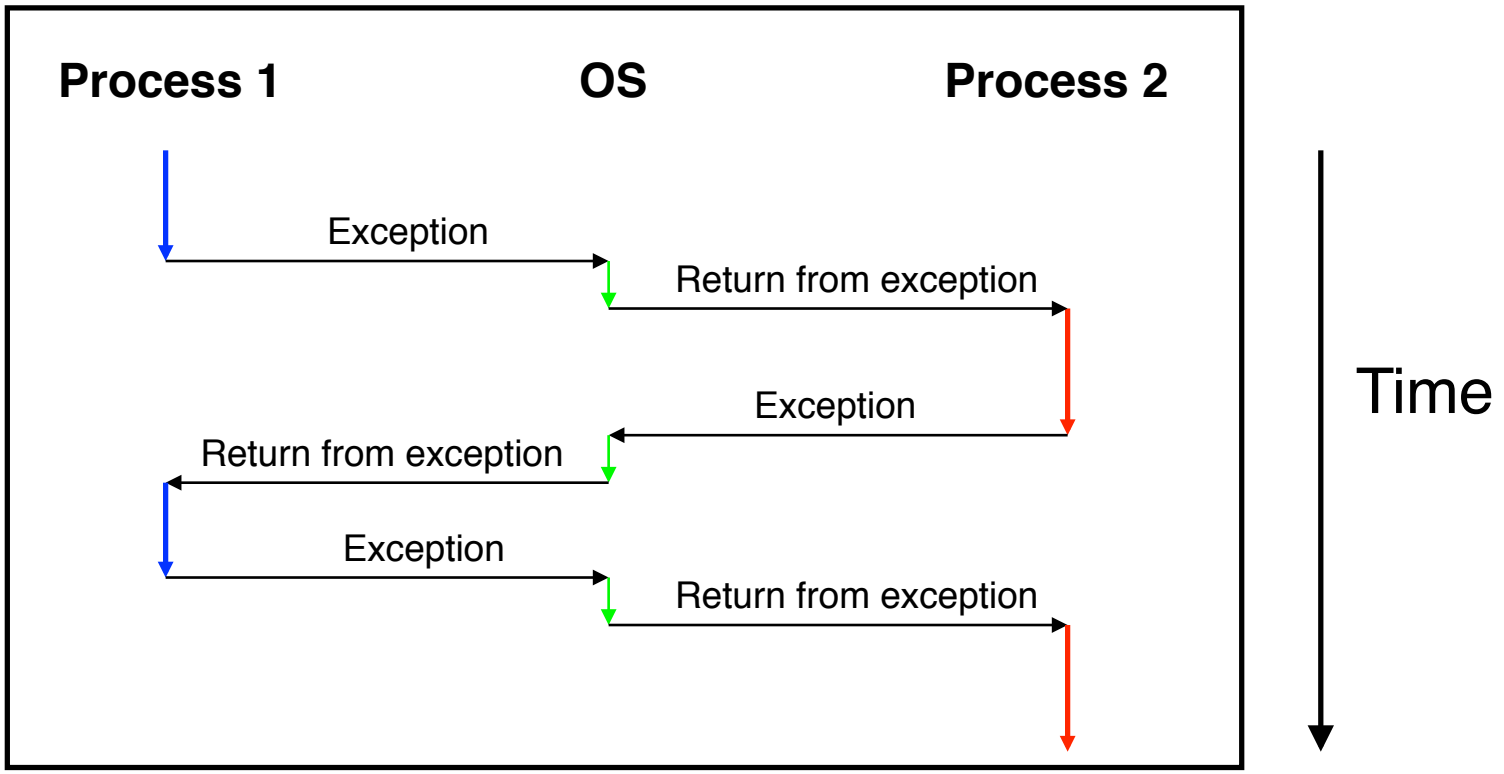
- **Process** is a profound abstraction in computer science
- The process abstraction provides application pgms with two key illusions:
  - Private control flow
  - Private address space

Private Control Flow: Illusion



Hardware and OS give each application process the illusion that it is the only process running on the CPU

Private Control Flow: Reality



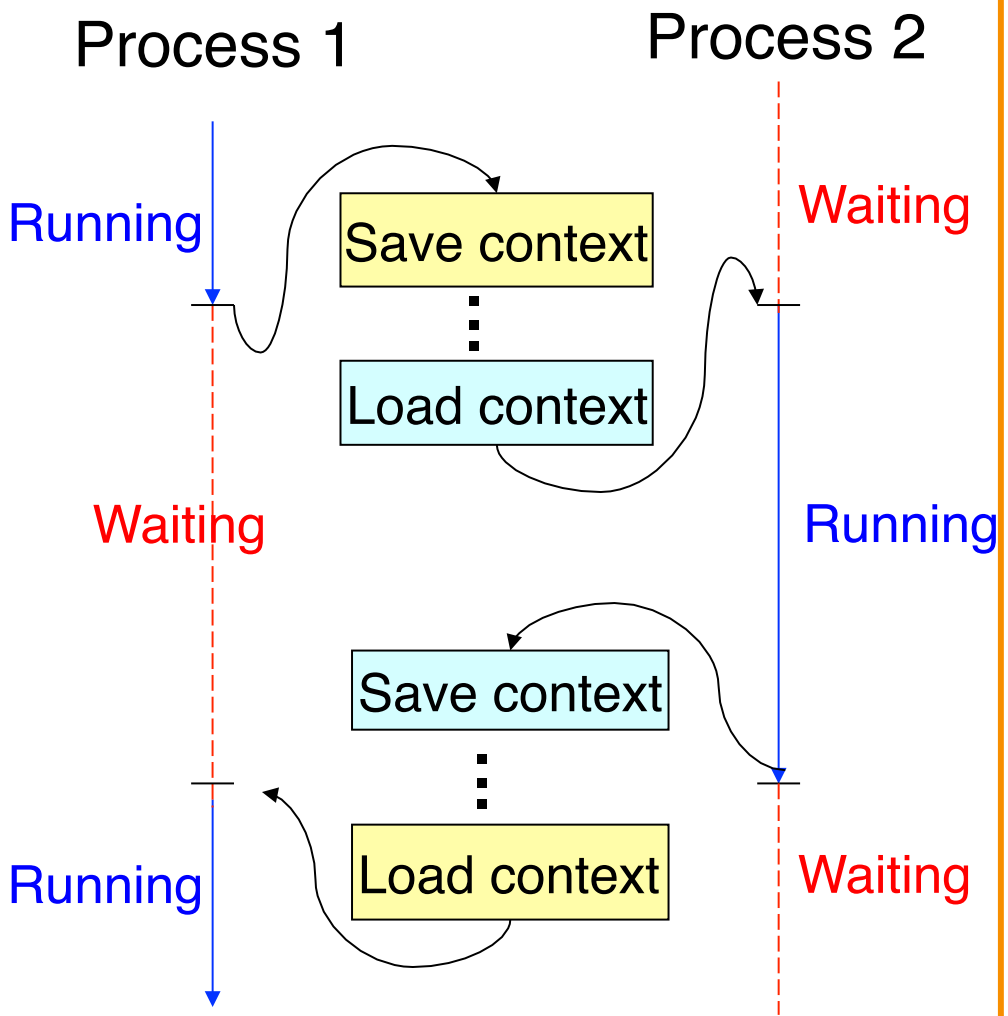
All application processes -- and the OS process -- share the same CPU(s)

Context Switches

- **Context switch**
  - The activity whereby the OS assigns the CPU to a different process
  - Occurs during exception handling, at discretion of OS
- Exceptions can be caused:
  - Synchronously, by application pgm (trap, fault, abort)
  - Asynchronously, by external event (interrupt)
  - **Asynchronously, by hardware timer**
    - So no process can dominate the CPUs
- Exceptions are the mechanism that enables the illusion of private control flow

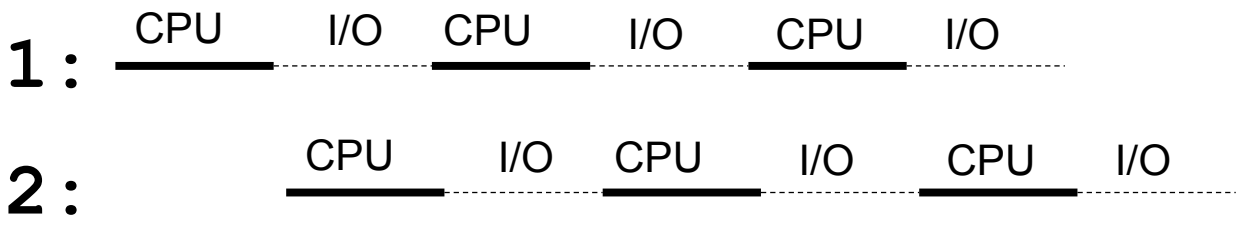
Context Switch Details

- Context
  - State the OS needs to restart a preempted process
- Context switch
  - Save the context of current process
  - Restore the saved context of some previously preempted process
  - Pass control to this newly restored process



When Should OS Do Context Switch?

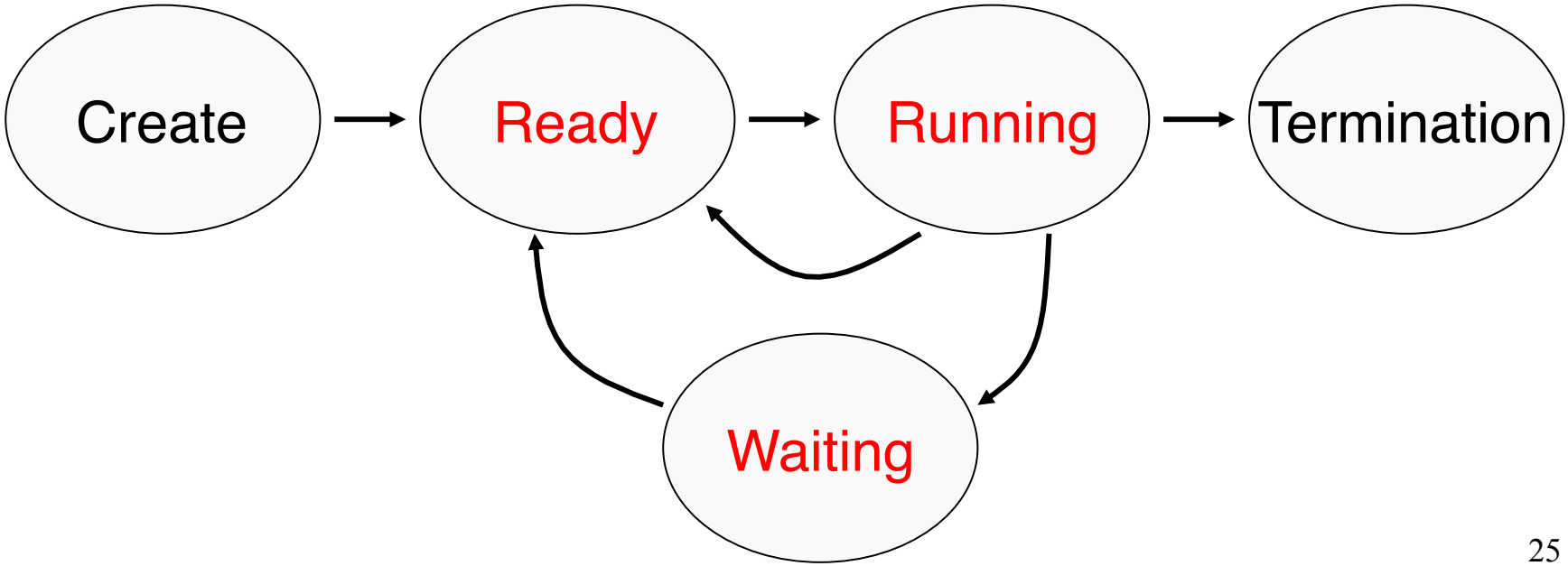
- When a process is stalled waiting for I/O
  - Better utilize the CPU, e.g., while waiting for disk access



- When a process has been running for a while
  - Sharing on a fine time scale to give each process the illusion of running on its own machine
  - Trade-off efficiency for a finer granularity of fairness

Life Cycle of a Process

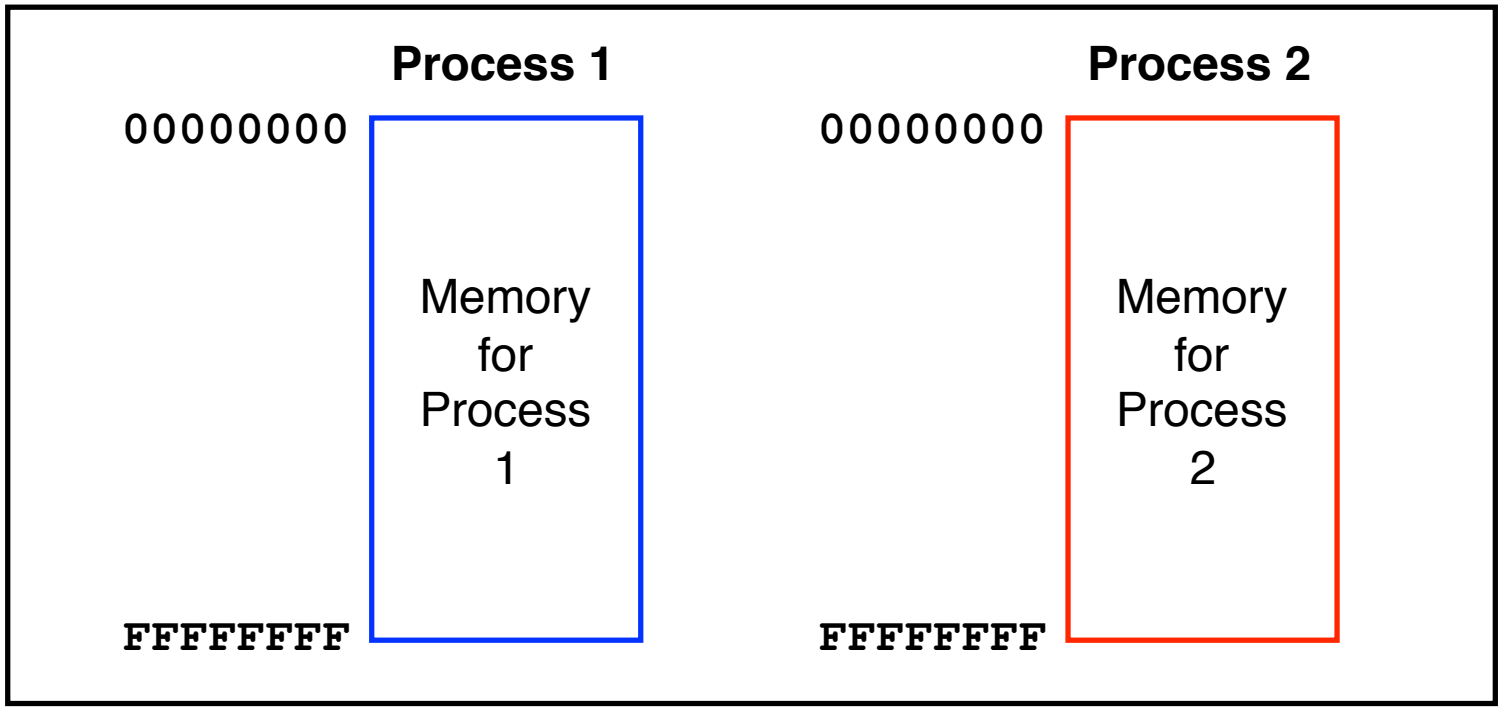
- **Running**: instructions are being executed
- **Waiting**: waiting for some event (e.g., I/O finish)
- **Ready**: ready to be assigned to a processor



Context Switch: What Context to Save?

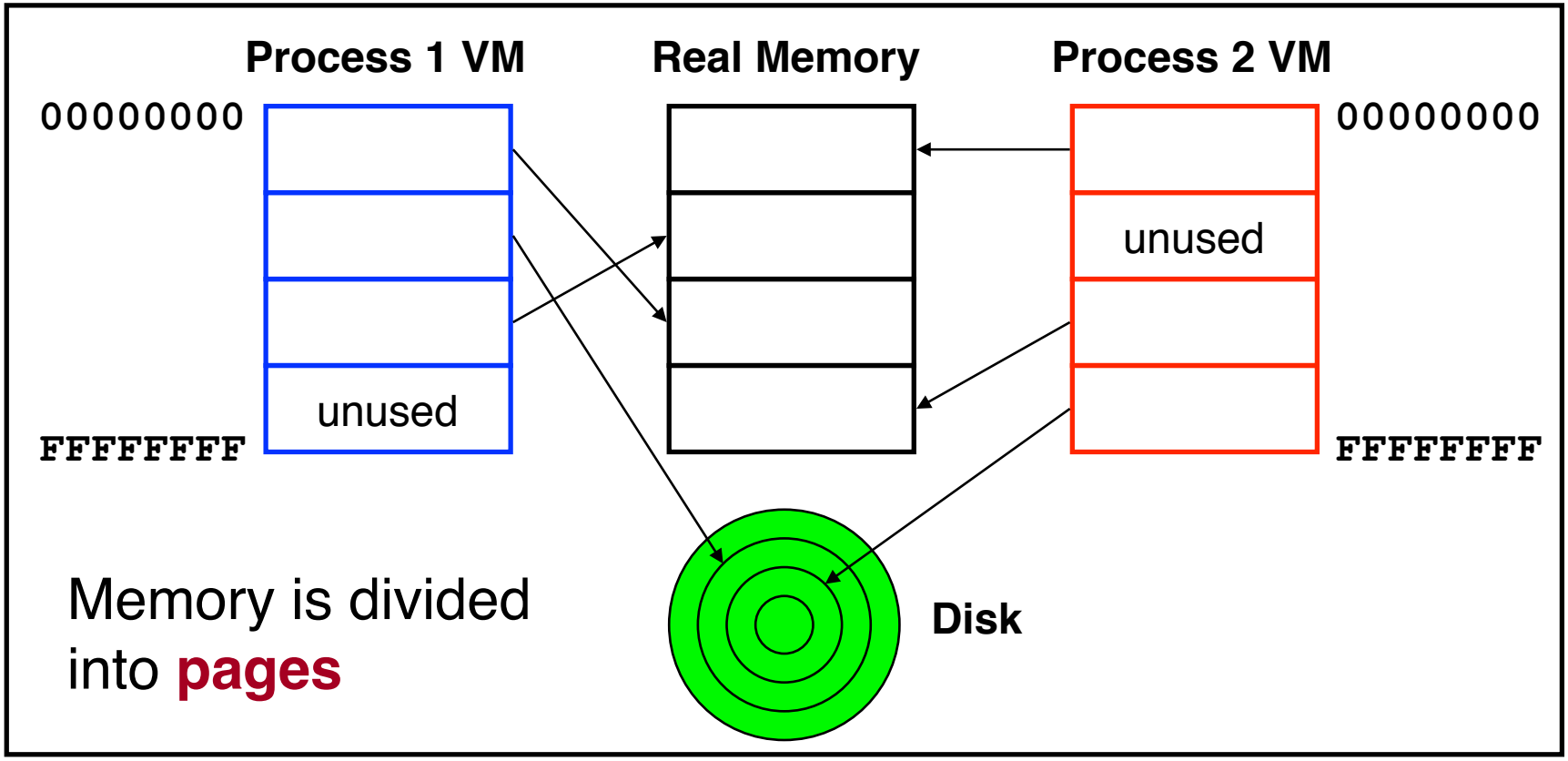
- Process state
  - New, ready, waiting, terminated
- CPU registers
  - EIP, EFLAGS, EAX, EBX, …
- I/O status information
  - Open files, I/O requests, …
- Memory management information
  - Page tables (see “Virtual Memory” lecture)
- Accounting information
  - Time limits, group ID, ...
- CPU scheduling information
  - Priority, queues

Private Address Space: Illusion



Hardware and OS give each application process the illusion that it is the only process using memory

Private Address Space: Reality



All processes use the same real memory
Hardware and OS provide application pgms with a **virtual** view of memory, i.e. **virtual memory (VM)**

Private Address Space Details

- Exceptions (specifically, page faults) are the mechanism that enables the illusion of private address spaces

- See the **Virtual Memory** lecture for details

Summary

- **Exception**: an abrupt change in control flow
  - **Interrupts**: asynchronous; e.g. I/O completion, hardware timer
  - **Traps**: synchronous; e.g. app pgm requests more heap memory, I/O
  - **Faults**: synchronous; e.g. seg fault
  - **Aborts**: synchronous; e.g. parity error
- **Process**: An instance of a program in execution
  - Hardware and OS use exceptions to give each process the illusion of:
    - Private control flow (reality: **context switches**)
    - Private address space (reality: **virtual memory**)

Appendix: System-Level Functions

Linux system-level functions for **I/O management**

Number	Function	Description
3	`read()`	Read data from file descriptor Called by `getchar()`, `scanf()`, etc.
4	`write()`	Write data to file descriptor Called by `putchar()`, `printf()`, etc.
5	`open()`	Open file or device Called by `fopen()`
6	`close()`	Close file descriptor Called by `fclose()`
8	`creat()`	Open file or device for writing Called by `fopen(…, "w")`

Described in **I/O Management** lecture

Appendix: System-Level Functions

Linux system-level functions for **process management**

Number	Function	Description
1	`exit()`	Terminate the process
2	`fork()`	Create a child process
7	`waitpid()`	Wait for process termination
7	`wait()`	(Variant of previous)
11	`exec()`	Execute a program in current process
20	`getpid()`	Get process id

Described in **Process Management** lecture

Appendix: System-Level Functions

Linux system-level functions for **I/O redirection** and **inter-process communication**

Number	Function	Description
41	`dup()`	Duplicate an open file descriptor
42	`pipe()`	Create a channel of communication between processes
63	`dup2()`	Close an open file descriptor, and duplicate an open file descriptor

Described in **Process Management** lecture

Appendix: System-Level Functions

Linux system-level functions for **dynamic memory management**

Number	Function	Description
45	`brk()`	Move the program break, thus changing the amount of memory allocated to the HEAP
45	`sbrk()`	(Variant of previous)
90	`mmap()`	Map a virtual memory page
91	`munmap()`	Unmap a virtual memory page

Described in **Dynamic Memory Management** lectures

Appendix: System-Level Functions

Linux system-level functions for **signal handling**

Number	Function	Description
27	`alarm()`	Deliver a signal to a process after a specified amount of wall-clock time
37	`kill()`	Send signal to a process
67	`sigaction()`	Install a signal handler
104	`setitimer()`	Deliver a signal to a process after a specified amount of CPU time
126	`sigprocmask()`	Block/unblock signals

Described in **Signals** lecture

Memory Management

Jennifer Rexford

Goals of this Lecture

- The memory hierarchy
- Spatial and temporal locality of reference
- Caching, at multiple levels
- Virtual memory

- How the hardware and OS give applications
  - The illusion of a large contiguous address space
  - Protection against each other

Virtual memory is one of the most important concepts in systems programming

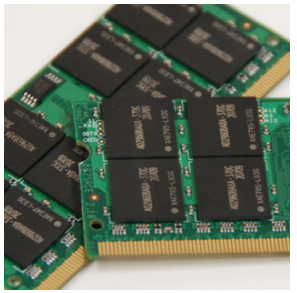
Motivation for Memory Hierarchy

- Faster storage technologies are more costly
  - Cost more money per byte
  - Have lower storage capacity
  - Require more power and generate more heat
- The gap between processing and memory is widening
  - Processors have been getting faster and faster
  - Main memory speed is not improving as dramatically
- Well-written programs tend to exhibit good locality
  - Across time: repeatedly referencing the same variables
  - Across space: often accessing other variables located nearby

Want the *speed* of fast storage at the *cost* and *capacity* of slow storage. Key idea: memory hierarchy!

Simple Three-Level Hierarchy

- Registers
  - Usually reside directly on the processor chip
  - Essentially no latency, referenced directly in instructions
  - Low capacity (e.g., 32-512 bytes)



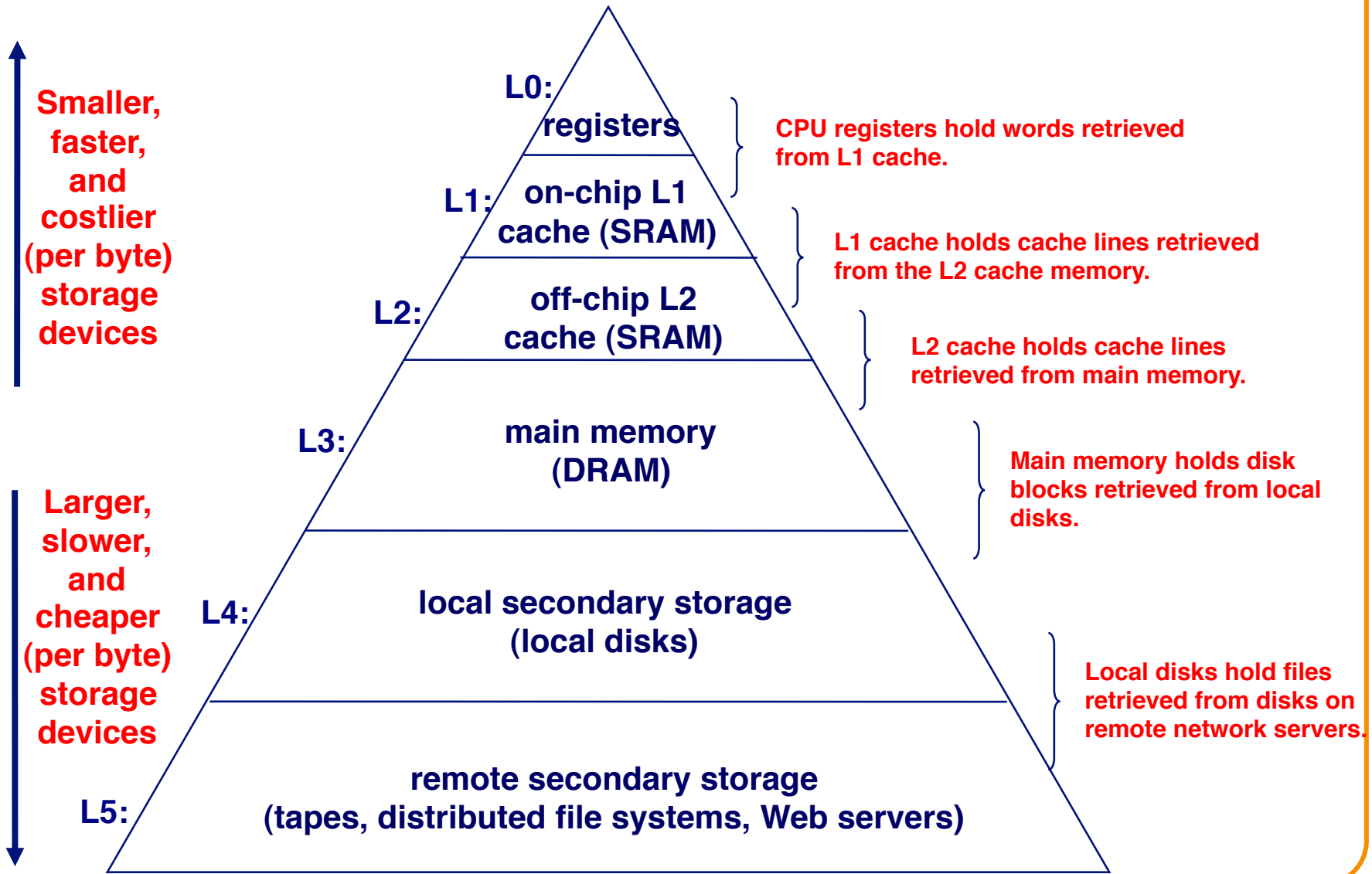
- Main memory
  - Around 100 times slower than a clock cycle
  - Constant access time for any memory location
  - Modest capacity (e.g., 512 MB-2GB)

- Disk
  - Around 100,000 times slower than main memory
  - Faster when accessing many bytes in a row
  - High capacity (e.g., 200 GB)

Widening Processor/Memory Gap

- Gap in speed increasing from 1986 to 2000
  - CPU speed improved ~55% per year
  - Main memory speed improved only ~10% per year
- Main memory as major performance bottleneck
  - Many programs stall waiting for reads and writes to finish
- Changes in the memory hierarchy
  - Increasing the number of registers
    - 8 integer registers in the x86 vs. 128 in the Itanium
  - Adding caches between registers and main memory
    - On-chip level-1 cache and off-chip level-2 cache

An Example Memory Hierarchy


Smaller, faster, and costlier (per byte) storage devices
Larger, slower, and cheaper (per byte) storage devices
L0:
registers
CPU registers hold words retrieved from L1 cache.
L1:
on-chip L1 cache (SRAM)
L1 cache holds cache lines retrieved from the L2 cache memory.
L2:
off-chip L2 cache (SRAM)
L2 cache holds cache lines retrieved from main memory.
L3:
main memory (DRAM)
Main memory holds disk blocks retrieved from local disks.
L4:
local secondary storage (local disks)
Local disks hold files retrieved from disks on remote network servers.
L5:
remote secondary storage (tapes, distributed file systems, Web servers)

Locality of Reference

- Two kinds of locality
  - **Temporal locality**: recently referenced items are likely to be referenced in near future
  - **Spatial locality**: Items with nearby addresses tend to be referenced close together in time.

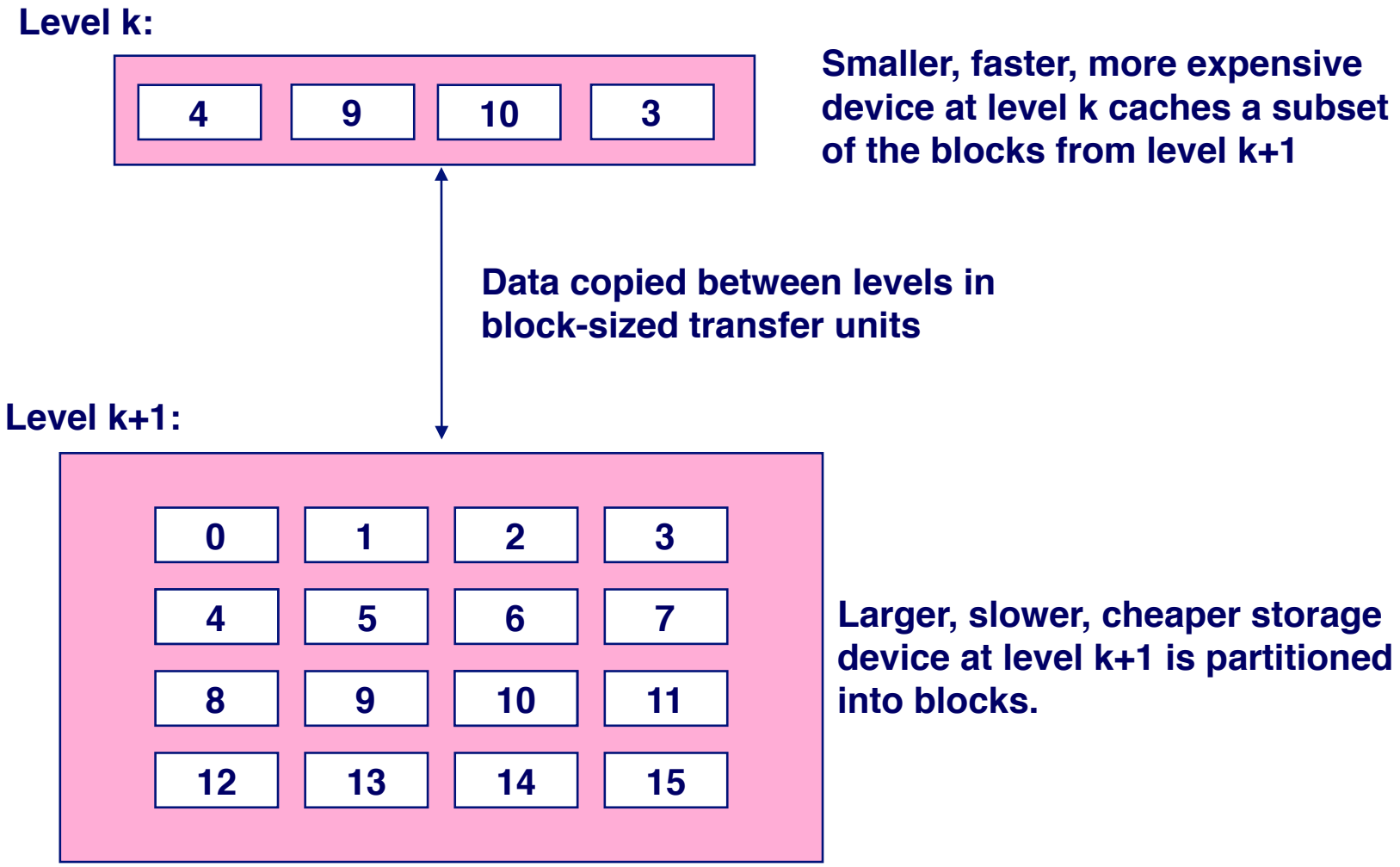
```
sum = 0;
for (i = 0; i < n; i++)
        sum += a[i];
return sum;
```

- Locality example
  - Program data
    - Temporal: the variable **`sum`**
    - Spatial: variable **`a[i+1]`** accessed soon after **`a[i]`**
  - Instructions
    - Temporal: cycle through the for-loop repeatedly
    - Spatial: reference instructions in sequence

Locality Makes Caching Effective

- Cache
  - Smaller, faster storage device that acts as a staging area
  - … for a *subset* of the data in a larger, slower device
- Caching and the memory hierarchy
  - Storage device at level k is a cache for level k+1
  - Registers as cache of L1/L2 cache and main memory
  - Main memory as a cache for the disk
  - Disk as a cache of files from remote storage
- Locality of access is the key
  - Most accesses satisfied by first few (faster) levels
  - Very few accesses go to the last few (slower) levels

Caching in a Memory Hierarchy

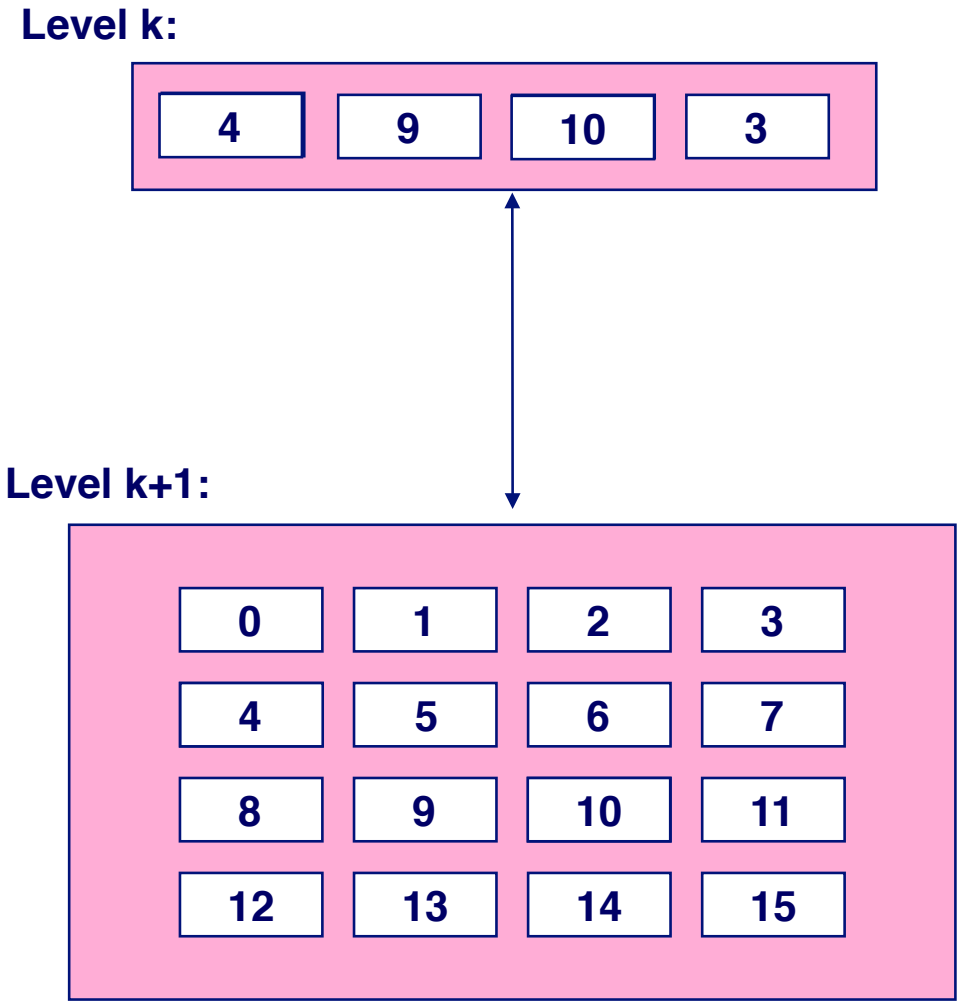


Cache Block Sizes

- Fixed vs. variable size
  - Fixed-sized blocks are easier to manage (common case)
  - Variable-sized blocks make more efficient use of storage
- Block size
  - Depends on access times at the level k+1 device
  - Larger block sizes further down in the hierarchy
  - E.g., disk seek times are slow, so disk pages are larger
- Examples
  - CPU registers: 4-byte words
  - L1/L2 cache: 32-byte blocks
  - Main memory: 4 KB pages
  - Disk: entire files

Cache Hit and Miss

- Cache hit
  - Program accesses a block available in the cache
  - Satisfy directly from cache
  - E.g., request for “10”
- Cache miss
  - Program accesses a block not available in the cache
  - Bring item into the cache
  - E.g., request for “13”
- Where to place the item?
- Which item to evict?



Three Kinds of Cache Misses

- Cold (compulsory) miss
  - Cold misses occur because the block hasn't been accessed before
  - E.g., first time a segment of code is executed
  - E.g., first time a particular array is referenced
- Capacity miss
  - Set of active cache blocks (the "working set") is larger than cache
  - E.g., manipulating a 1200-byte array within a 1000-byte cache
- Conflict miss
  - Some caches limit the locations where a block can be stored
  - E.g., block i must be placed in cache location (i mod 4)
  - Conflicts occur when multiple blocks map to the same location(s)
  - E.g., referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time

Cache Replacement

- Evicting a block from the cache
  - New block must be brought into the cache
  - Must choose a "victim" to evict
- Optimal eviction policy
  - Evict a block that is *never* accessed again
  - Evict the block accessed the *furthest in the future*
  - Impossible to implement without knowledge of the future
- Using the past to predict the future
  - Evict the "least recently used" (LRU) block
  - Assuming it is not likely to be used again soon
- But, LRU is often expensive to implement
  - Need to keep track of access times
  - So, simpler approximations of LRU are used

Who Manages the Cache?

- Registers
  - Cache of L1/L2 cache and main memory
  - Managed explicitly by the *compiler*
  - By determining which data are brought in and out of registers
  - Using relatively sophisticated code-analysis techniques
- L1/L2 cache
  - Cache of main memory
  - Managed by the *hardware*
  - Using relatively simple mechanisms (e.g., “i mod 4”)
- **Main memory**
  - **Cache of the disk**
  - **Managed (in modern times) by the *operating system***
  - **Using relatively sophisticated mechanisms (e.g., LRU-like)**
  - **Since reading from disk is extremely time consuming**

Manual Allocation: Segmentation

- In the olden days (aka "before the mid 1950s")
  - Programmers incorporated storage allocation in their programs
  - … whenever the total information exceeded main memory
- Segmentation
  - Programmers would divide their programs into "segments"
  - Which would "overlay" (i.e., replace) one another in main memory
- Advantages
  - Programmers are intimately familiar with their code
  - And can optimize the layout of information in main memory
- Disadvantages
  - Immensely tedious and error-prone
  - Compromises the portability of the code

Automatic Allocation: Virtual Memory

- Give programmer the illusion of a very large memory
  - Large: 4 GB of memory with 32-bit addresses
  - Uniform: contiguous memory locations, from 0 to 2^{32}-1
- Independent of
  - The actual size of the main memory
  - The presence of any other processes sharing the computer
- Key idea #1: separate "address" from "physical location"
  - Virtual addresses: generated by the program
  - Memory locations: determined by the hardware and OS
- Key idea #2: caching
  - Swap virtual pages between main memory and the disk

One of the greatest ideas in computer systems!

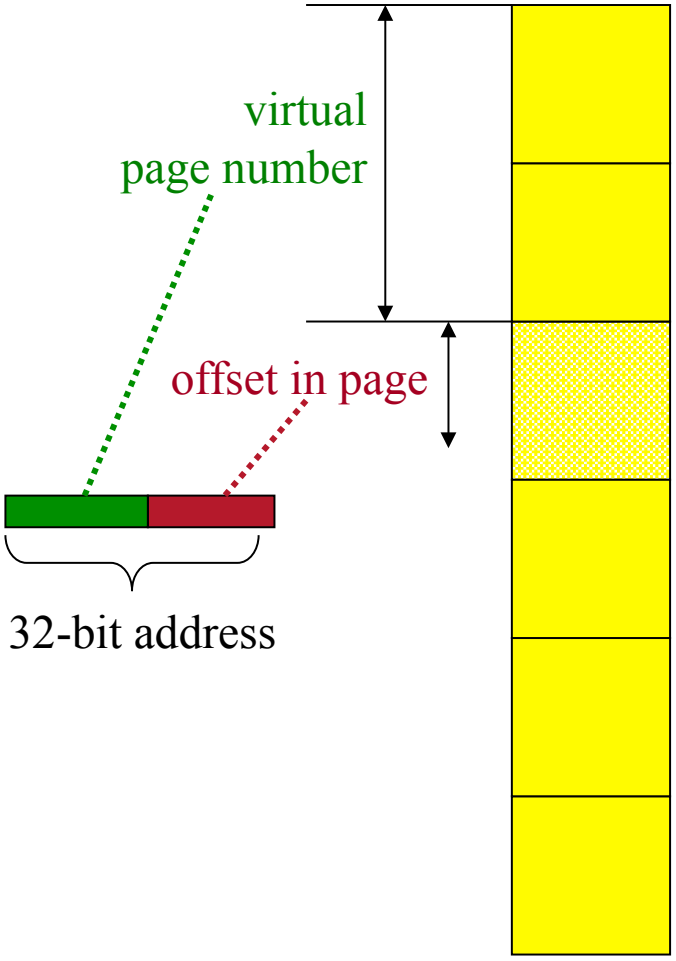
Making Good Use of Memory and Disk

- Good use of the disk
  - Read and write data in large "pages"
  - … to amortize the cost of "seeking" on the disk
  - E.g., page size of 4 KB
- Good use of main memory
  - Even though the address space is large
  - … programs usually access only small portions at a time
  - Keep the "working set" in main memory
    - Demand paging: only bring in a page when needed
    - Page replacement: selecting good page to swap out
- Goal: avoid thrashing
  - Continually swapping between memory and disk

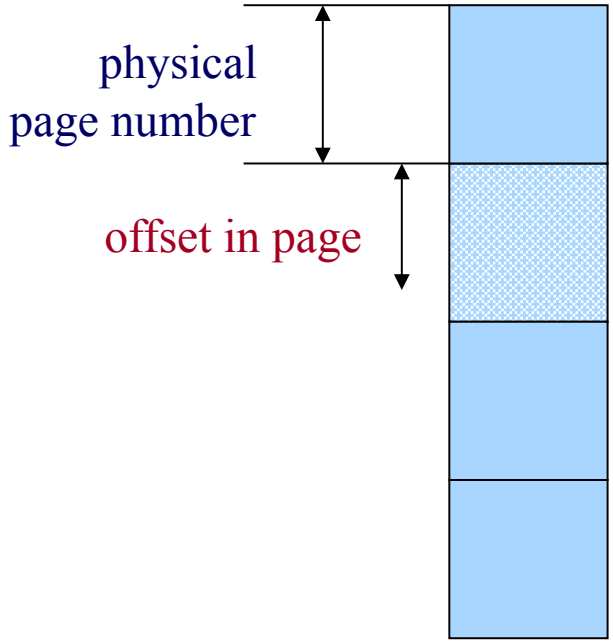
Virtual Address for a Process

- Virtual page number
  - Number of the page in the virtual address space
  - Extracted from the upper bits of the (virtual) address
  - … and then mapped to a physical page number
- Offset in a page
  - Number of the byte within the page
  - Extracted from the lower bits of the (virtual) address
  - … and then used as offset from start of physical page
- Example: 4 KB pages
  - 20-bit page number: 2^{20} virtual pages
  - 12-bit offset: bytes 0 to 2^{12}-1

Virtual Memory for a Process



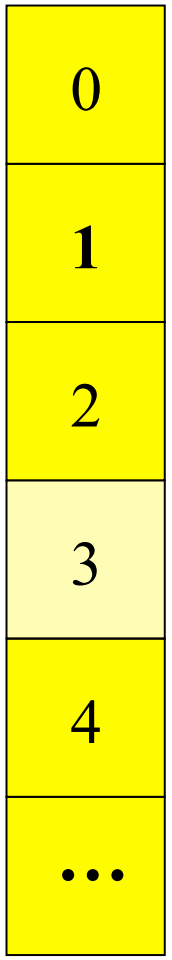
Translate virtual page number
to physical page number

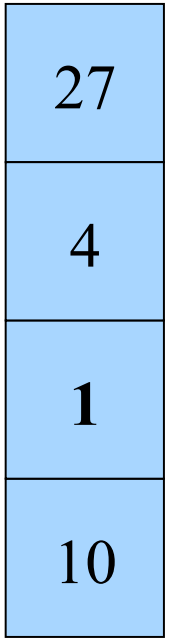


Virtual Address Space Physical Address Space

Page Table to Manage the Cache

- Current location of each virtual page
  - Physical page number, or
  - Disk address (or null if unallocated)
- Example
  - Page 0: at location xx on disk
  - Page 1: at physical page 2
  - Page 3: not yet allocated
- Page "hit" handled by hardware
  - Compute the physical address
    - Map virtual page # to physical page #
    - Concatenate with offset in page
  - Read or write from main memory
    - Using the physical address
- Page "miss" triggers an exception…





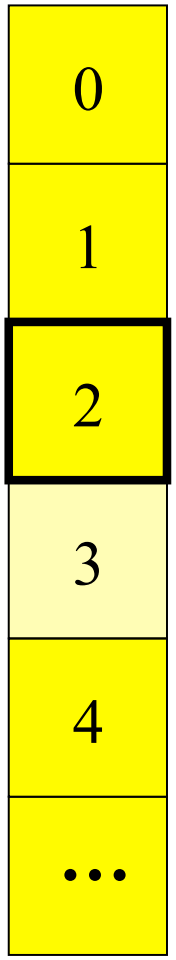
“Miss” Triggers Page Fault

- Accessing page not in main memory

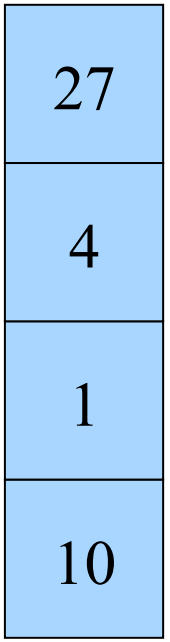
	V	Physical or disk address
0	0	xx
1	1	2
2	**0**	**yy**
3	0	null
4	1	1
		…

```
movl   0002104, %eax
```

Virtual page #2 at location yy on disk!



virtual pages

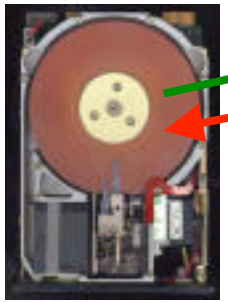


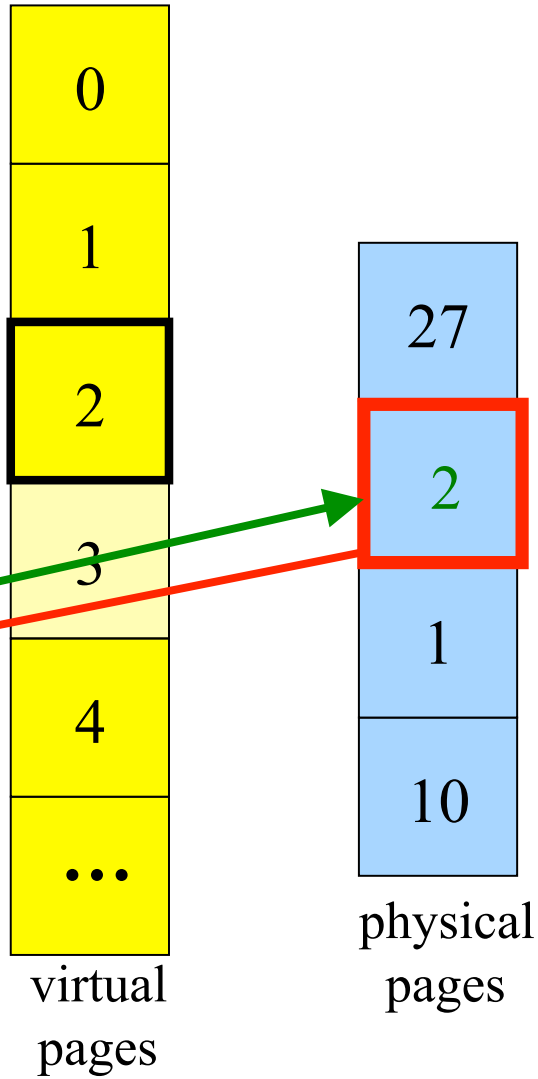
physical pages

OS Handles the Page Fault

- Bringing page in from disk
  - If needed, swap out old page (e.g., #4)
  - Bring in the new page (page #2)
  - Update the page table entries

	V	Physical or disk address
0	0	xx
1	1	2
2	~~0~~ 1	~~yy~~ 1
3	0	null
4	~~1~~ 0	~~1~~ zz
		…

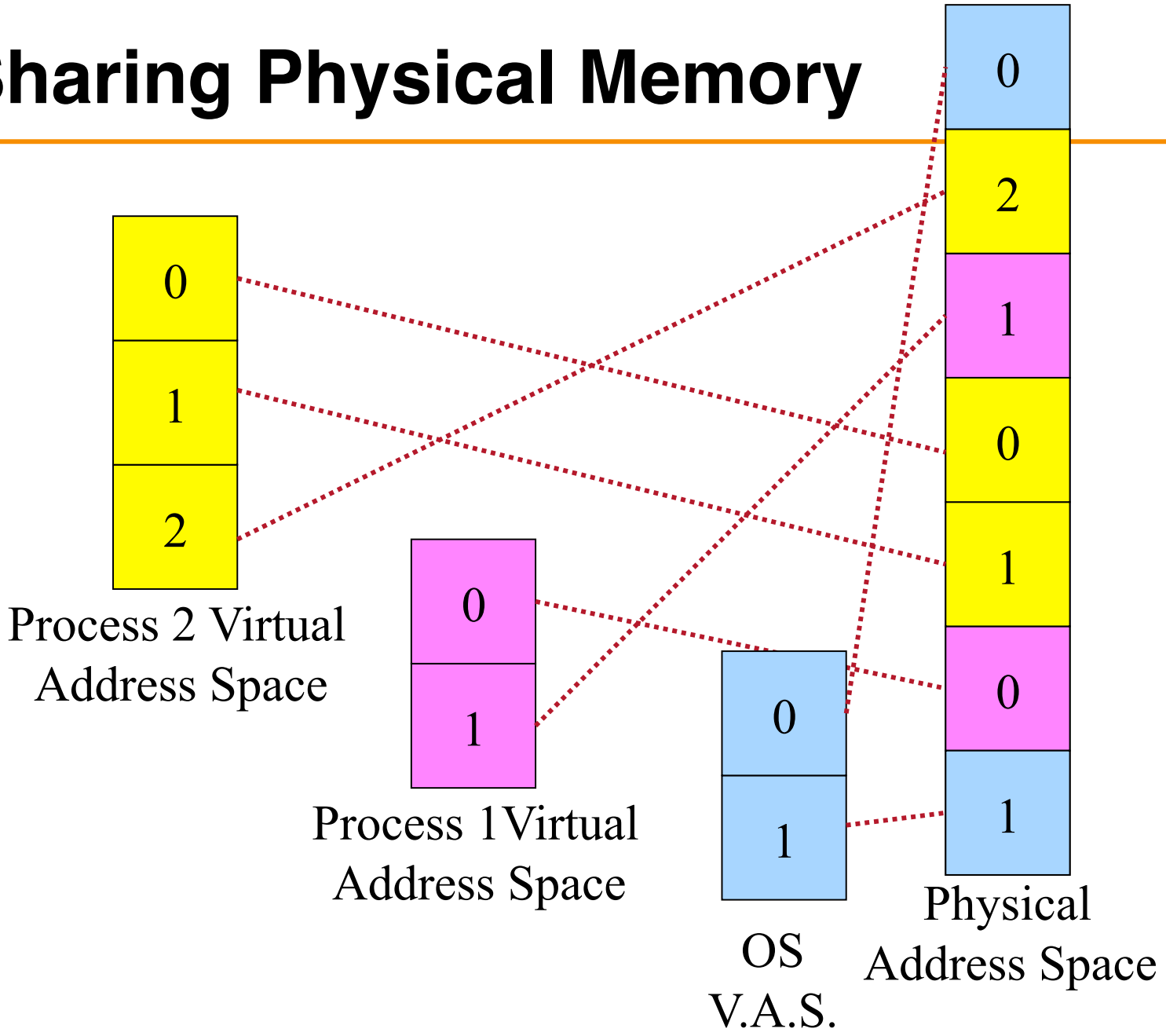




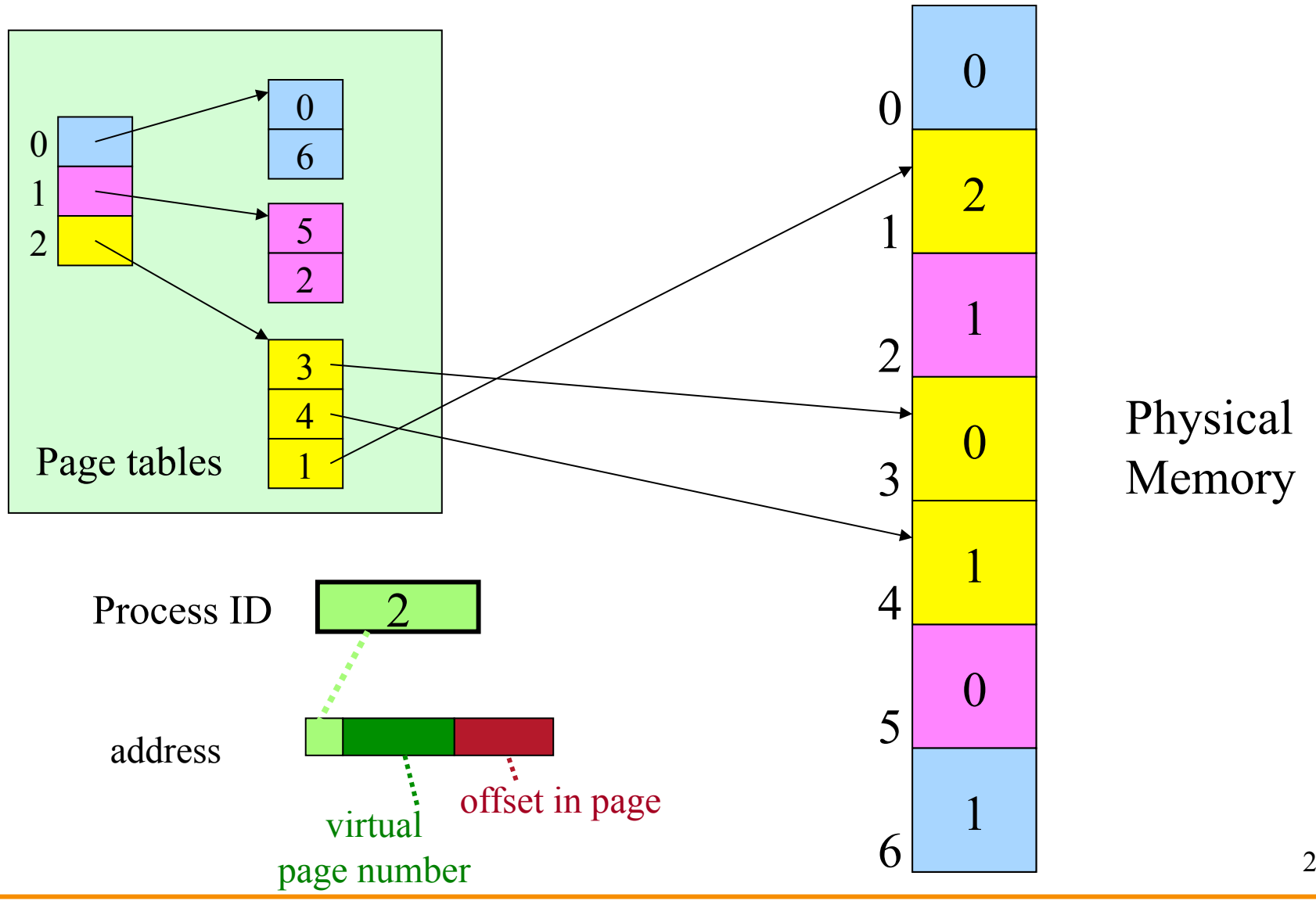
VM as a Tool for Memory Protection

- Memory protection
  - Prevent process from unauthorized reading or writing of memory
- User process should not be able to
  - Modify the read-only text section in its own address space
  - Read or write operating-system code and data structures
  - Read or write the private memory of other processes
- Hardware support
  - Permission bits in page-table entries (e.g., read-only)
  - Separate identifier for each process (i.e., process-id)
  - Switching between *unprivileged* mode (for user processes) and *privileged* mode (for the operating system)

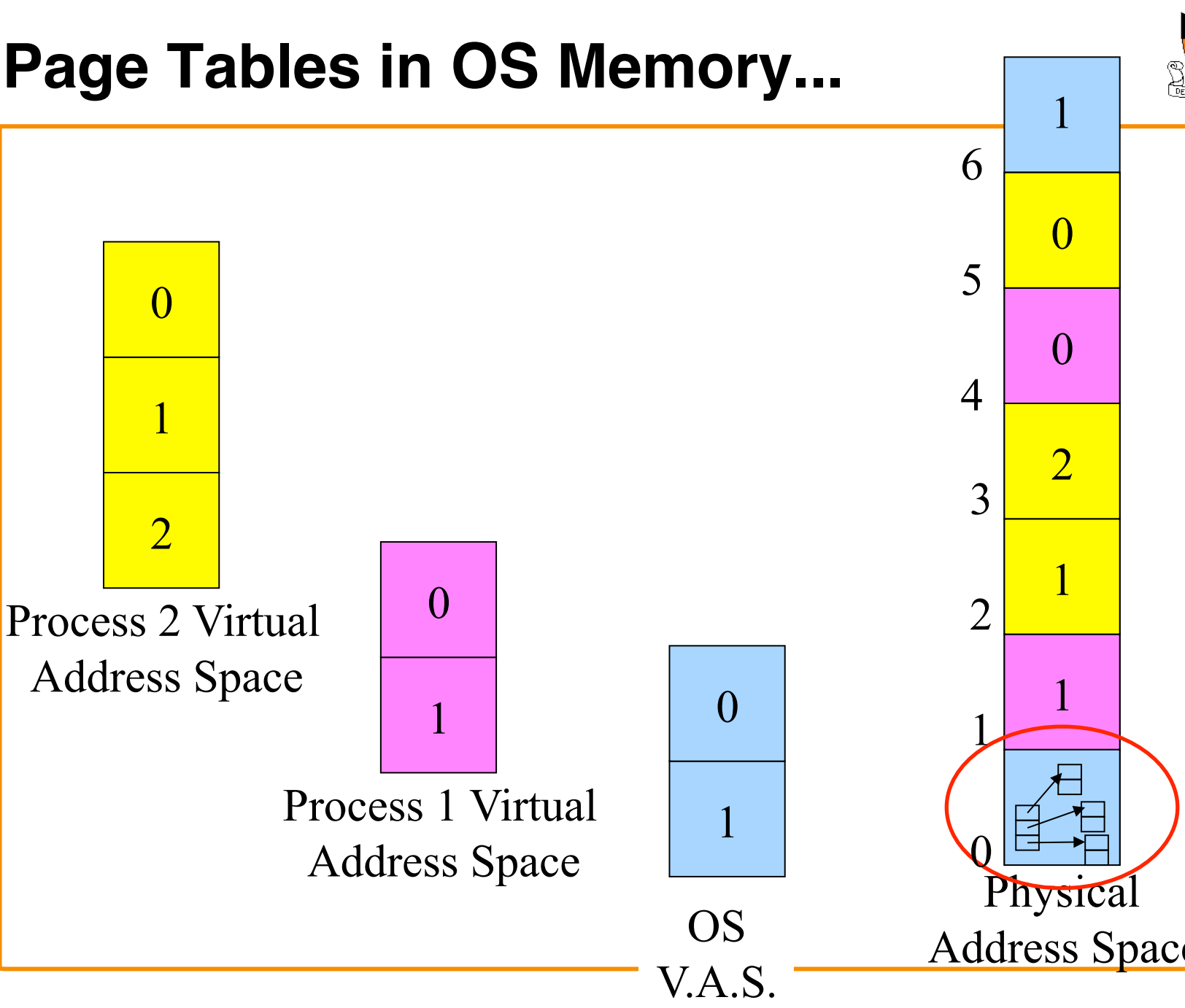
Sharing Physical Memory



Process-ID and Page Table Entries



Page Tables in OS Memory...

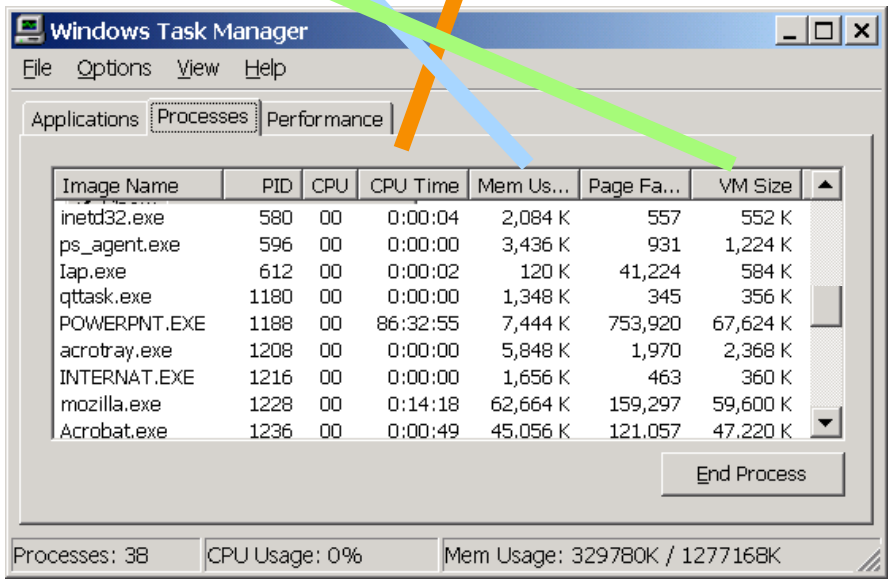


Measuring the Memory Usage

Virtual memory usage
Physical memory usage ("resident set size")
CPU time used by this process so far

```
% ps l
F   UID   PID  PPID PRI    VSZ    RSS STAT  TIME COMMAND
0   115  7264  7262  17   4716   1400   SN  0:00 -csh
0   115  7290  7264  17  15380  10940   SN  5:52 emacs
0   115  3283  7264  23   2864    812   RN  0:00 ps l
```

Unix


Windows Task Manager
File Options View Help
Applications Processes Performance
Image Name PID CPU CPU Time Mem Us... Page Fa... VM Size
inetd32.exe 580 00 0:00:04 2,084 K 557 552 K
ps_agent.exe 596 00 0:00:00 3,436 K 931 1,224 K
Iap.exe 612 00 0:00:02 120 K 41,224 584 K
qttask.exe 1180 00 0:00:00 1,348 K 345 356 K
POWERPNT.EXE 1188 00 86:32:55 7,444 K 753,920 67,624 K
acrotray.exe 1208 00 0:00:00 5,848 K 1,970 2,368 K
INTERNAT.EXE 1216 00 0:00:00 1,656 K 463 360 K
mozilla.exe 1228 00 0:14:18 62,664 K 159,297 59,600 K
Acrobat.exe 1236 00 0:00:49 45,056 K 121,057 47,220 K
End Process
Processes: 38
CPU Usage: 0%
Mem Usage: 329780K / 1277168K

Windows

VM as a Tool for Memory Management

- Simplifying linking
  - Same memory layout for each process
    - E.g., text section always starts at **`0x08048000`**
    - E.g., stack always grows down from **`0xbfffffff`**
  - Linker can be independent of physical location of code
- Simplifying sharing
  - User processes can share some code and data
    - E.g., single physical copy of stdio library code (like printf)
  - Mapped in to the virtual address space of each process
- Simplifying memory allocation
  - User processes can request additional memory from the heap
    - E.g., using **`malloc()`** to allocate, and **`free()`** to deallocate
  - OS allocates *contiguous* virtual pages…
    - … and scatters them *anywhere* in physical memory

Summary

- Memory hierarchy
  - Memory devices of different speed, size, and cost
  - Registers, on-chip cache, off-chip cache, main memory, disk, tape
  - Locality of memory accesses making caching effective
- Virtual memory
  - Separate virtual address space for each process
  - Provides caching, memory protection, and memory management
  - Implemented via cooperation of the address-translation hardware and the OS (when page faults occur)
- In **Dynamic Memory Management** lectures:
  - Dynamic memory allocation on the heap
  - Management by user-space software (e.g., **`malloc()`** and **`free()`**)

Dynamic Memory Management

Jennifer Rexford

Goals of this Lecture

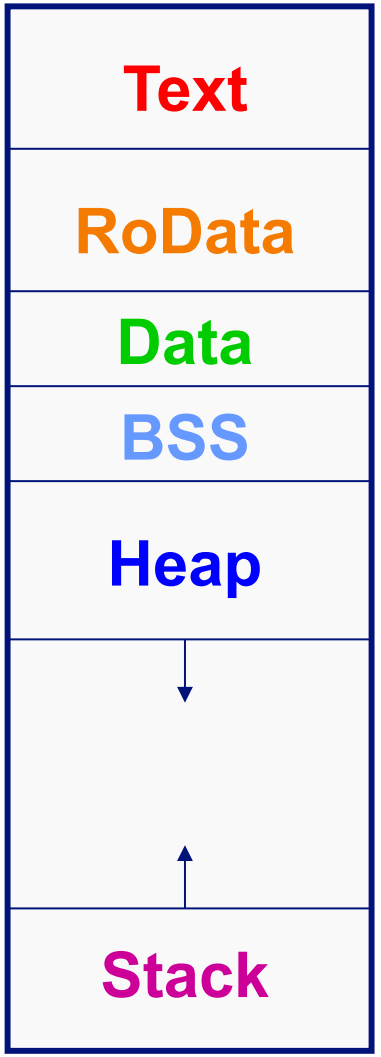
- Dynamic memory management techniques
  - Garbage collection by the run-time system (Java)
  - Manual de-allocation by the programmer (C, C++)
- Design decisions for the "K&R" heap manager
  - Circular linked-list of free blocks with a "first fit" allocation
  - Coalescing of adjacent blocks to create larger blocks

Part 1:

What do `malloc()` and `free()` do?

Memory Layout: Heap

```
char* string = "hello";
int iSize;

char* f()
{
    char* p;
    scanf("%d", &iSize);
    p = malloc(iSize);
    return p;
}
```



Needed when required memory size is not known before the program runs

Allocating & Deallocating Memory

- Dynamically *allocating* memory
  - Programmer explicitly requests space in memory
  - Space is allocated dynamically on the heap
  - E.g., using “malloc” in C, and “new” in Java
- Dynamically *deallocating* memory
  - Must reclaim or recycle memory that is never used again
  - To avoid (eventually) running out of memory
- “Garbage”
  - Allocated block in heap that will not be accessed again
  - Can be reclaimed for later use by the program

Option #1: Garbage Collection

- *Run-time system* does garbage collection (Java)
  - Automatically determines objects that can't be accessed
  - And then reclaims the resources used by these objects

```
Object x = new Foo();
Object y = new Bar();
x = new Quux();

if (x.check_something()) {
  x.do_something(y);
}
System.exit(0);
```

Object `Foo()` is never used again!

Challenges of Garbage Collection

- Detecting the garbage is not always easy
  - "if (complex_function(y)) x = Quux();"
  - Run-time system cannot collect *all* of the garbage
- Detecting the garbage introduces overhead
  - Keeping track of references to objects (e.g., counter)
  - Scanning through accessible objects to identify garbage
  - Sometimes walking through a large amount of memory
- Cleaning the garbage leads to bursty delays
  - E.g., periodic scans of the objects to hunt for garbage
  - Leading to unpredictable "freeze" of the running program
  - Very problematic for real-time applications
  - … though good run-time systems avoid long freezes

Option #2: Manual Deallocation

- *Programmer* deallocates the memory (C and C++)
  - Manually determines which objects can't be accessed
  - And then explicitly returns the resources to the heap
  - E.g., using "free" in C or "delete" in C++
- Advantages
  - Lower overhead
  - No unexpected "pauses"
  - More efficient use of memory
- Disadvantages
  - More complex for the programmer
  - Subtle memory-related bugs
  - Security vulnerabilities in the (buggy) code

Manual Deallocation Can Lead to Bugs

- Dangling pointers
  - Programmer frees a region of memory
  - … but still has a pointer to it
  - Dereferencing pointer reads or writes *nonsense values*

```
int main(void) {
    char *p;
    p = malloc(10);
    …
    free(p);
    …
    putchar(*p);
}
```

May print nonsense character.

Manual Deallocation Can Lead to Bugs

- Memory leak
  - Programmer neglects to free unused region of memory
  - So, the space can never be allocated again
  - Eventually may consume all of the available memory

```
void f(void) {
    char *s;
    s = malloc(50);
    return;
}

int main(void) {
    while (1) f();
    return 0;
}
```

Eventually, malloc() returns NULL

Manual Deallocation Can Lead to Bugs

- Double free
  - Programmer mistakenly frees a region more than once
  - Leading to corruption of the heap data structure
  - … or premature destruction of a *different* object

```
int main(void) {
    char *p, *q;
    p = malloc(10);
    …
    free(p);
    q = malloc(10);
    free(p);
    …
}
```

Might free the space allocated to **q**!

`malloc()` and `free()` Challenges

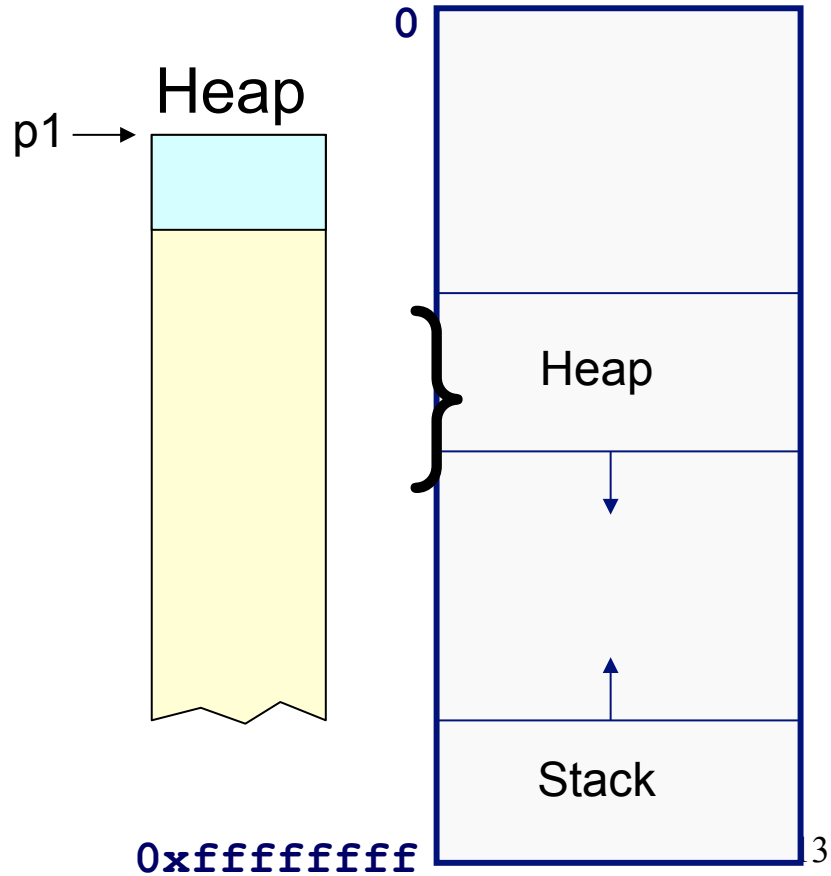
- **`malloc()`** may ask for arbitrary number of bytes
- Memory may be allocated & freed in different order
- Cannot reorder requests to improve performance

```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```

Heap: Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

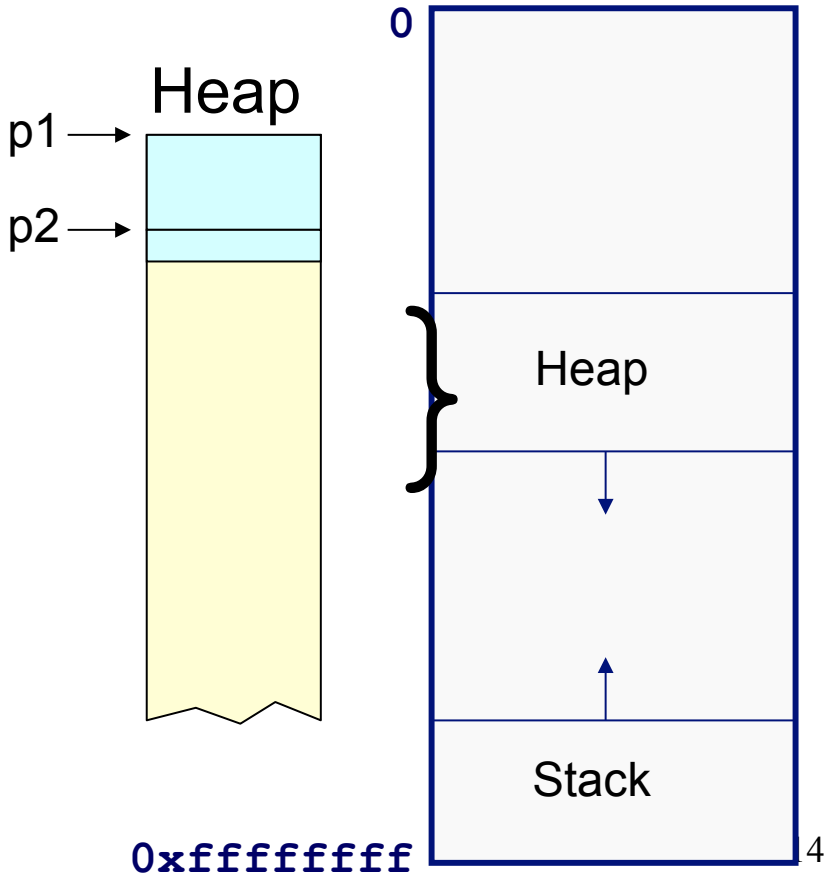
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Heap: Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

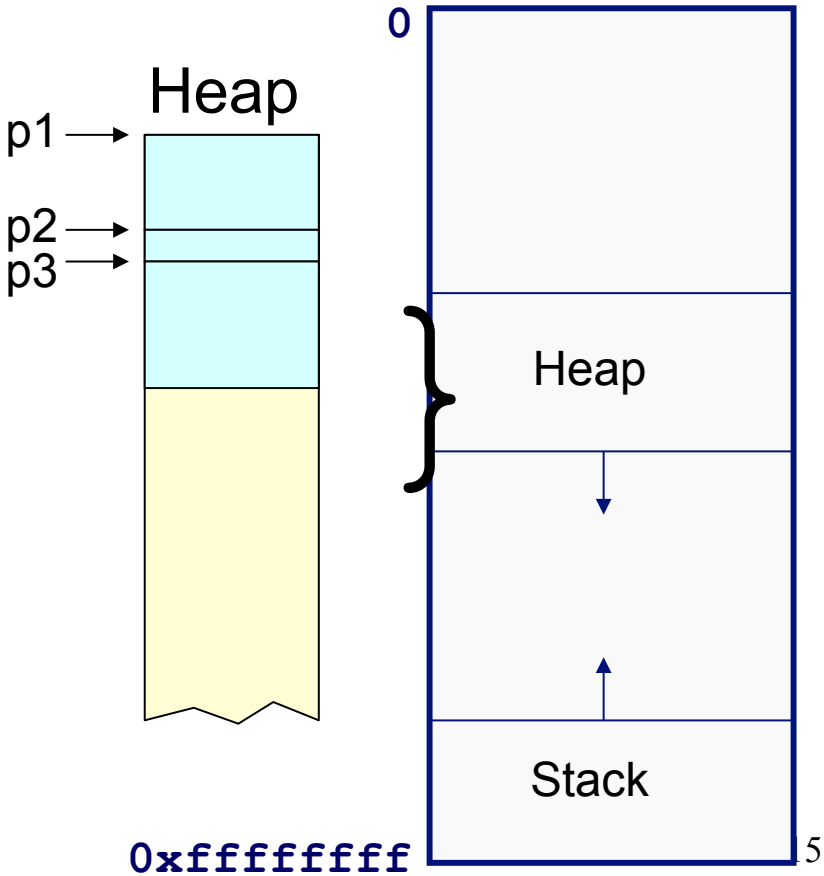
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Heap: Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

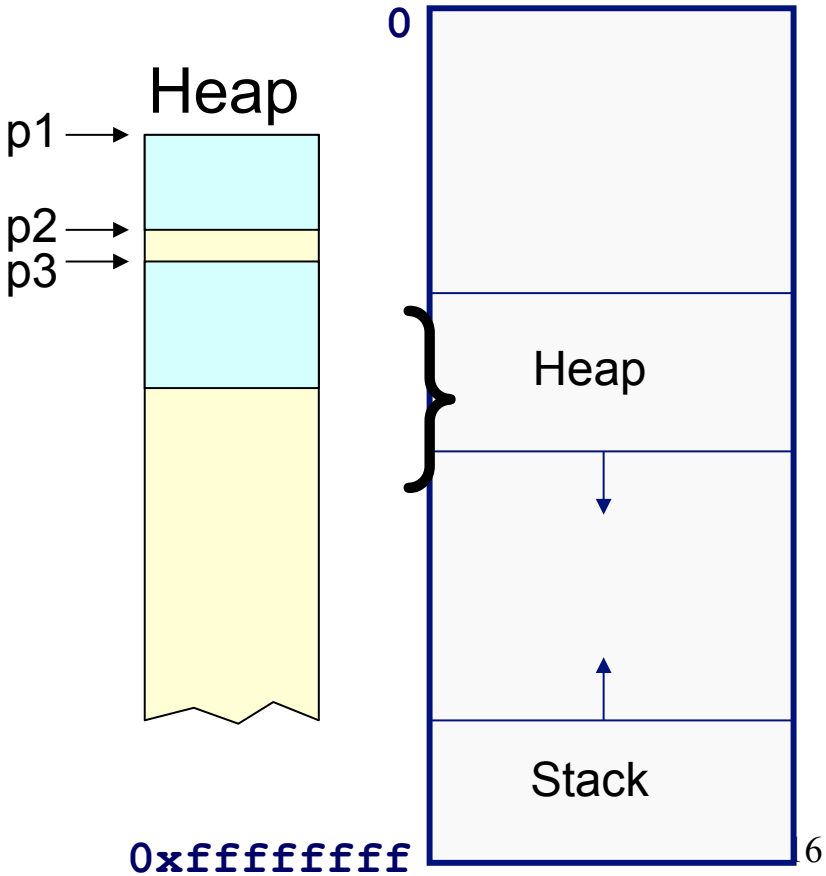
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Heap: Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

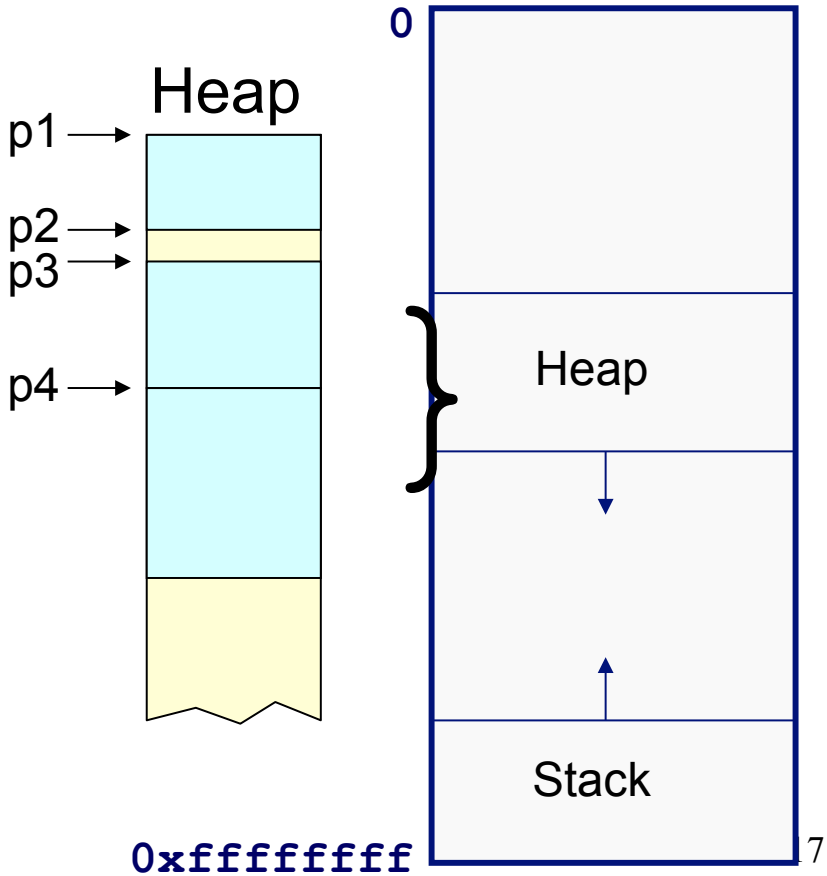
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Heap: Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

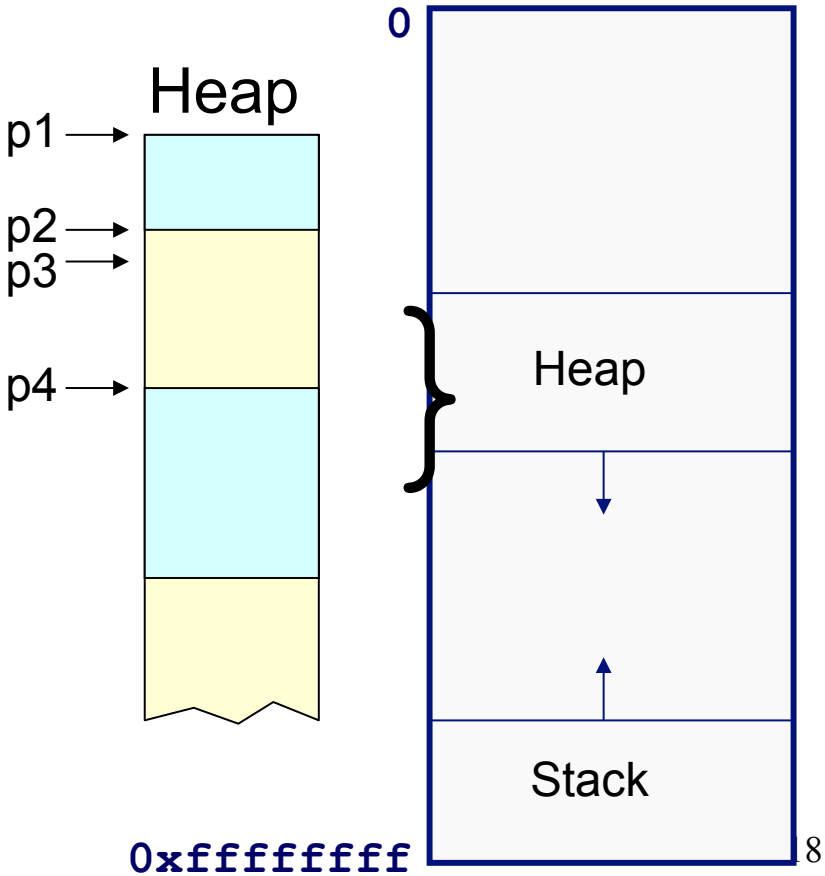
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Heap: Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

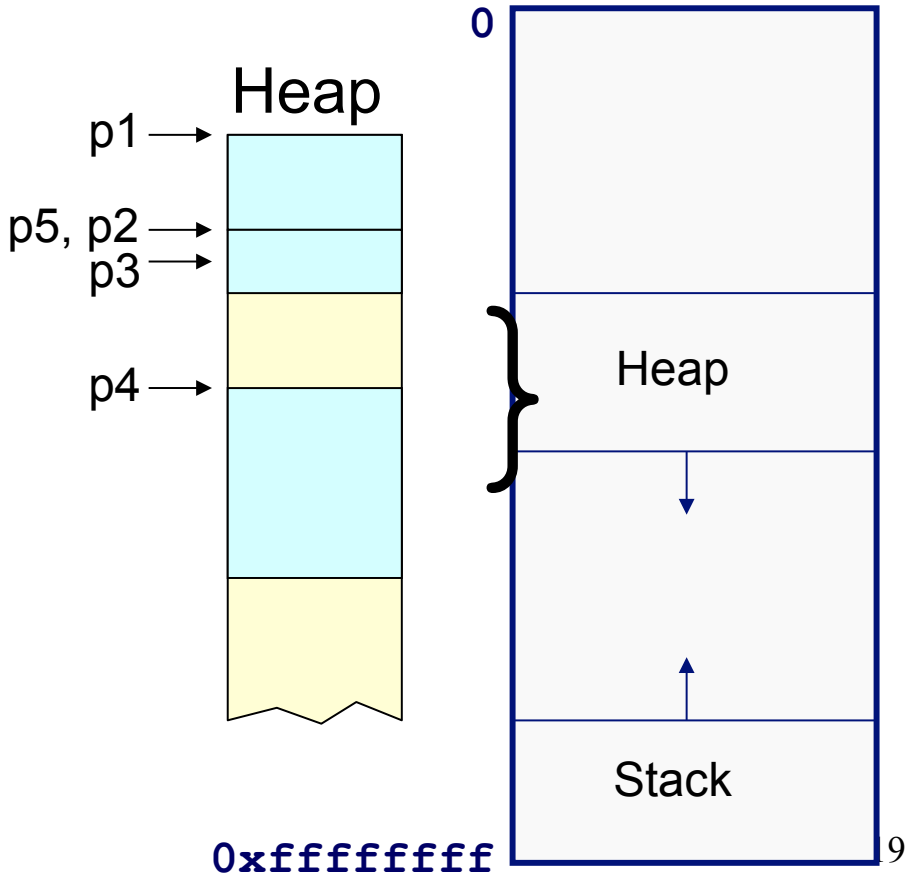
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Heap: Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

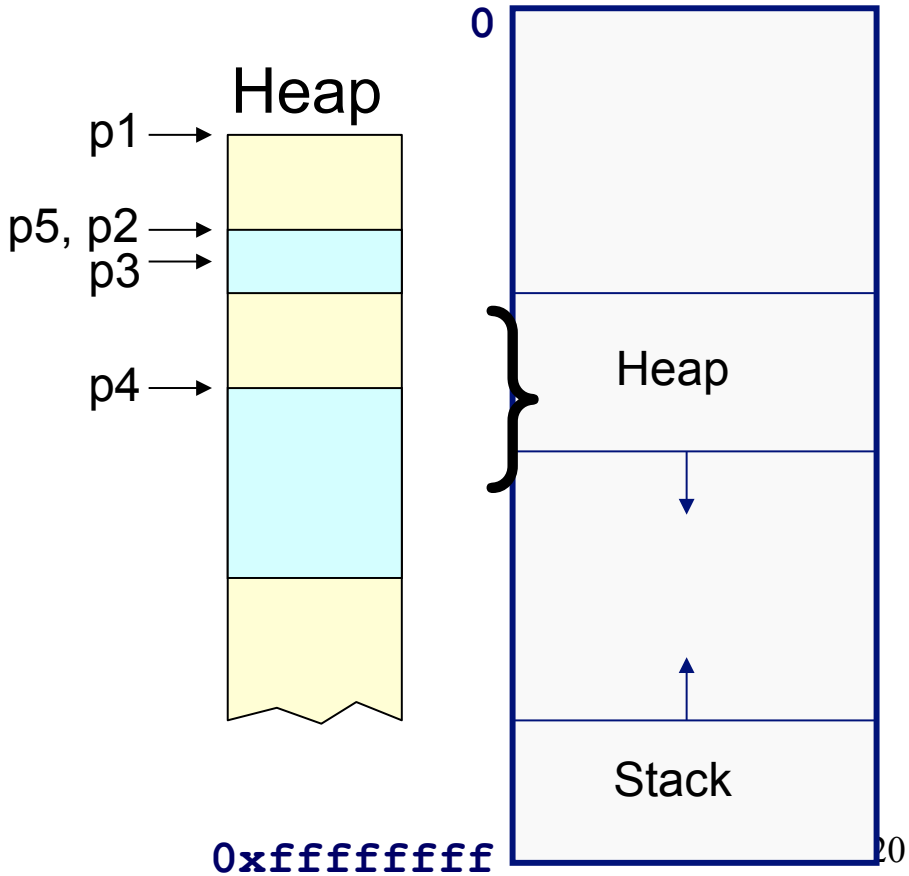
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Heap: Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

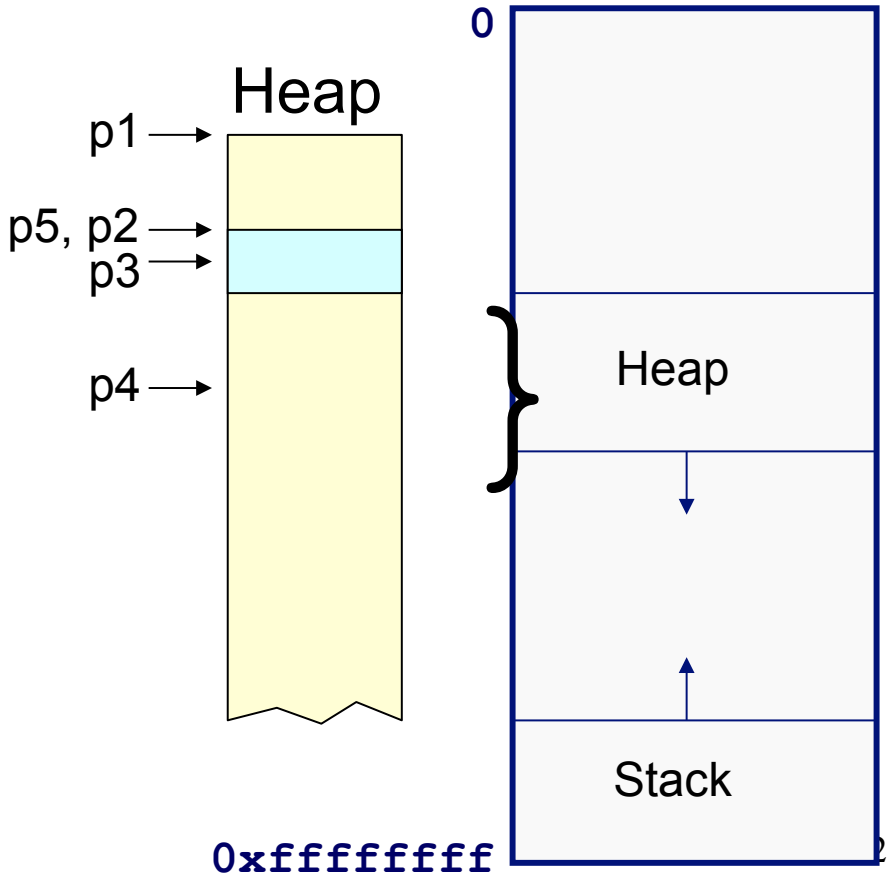
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Heap: Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

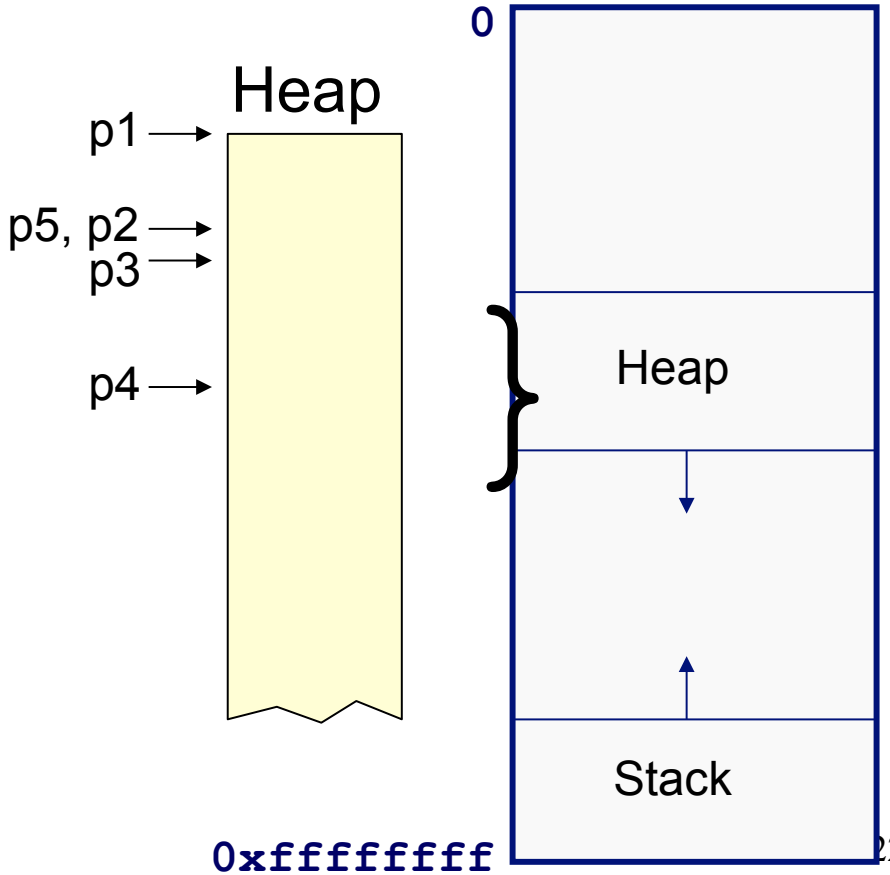
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Heap: Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```

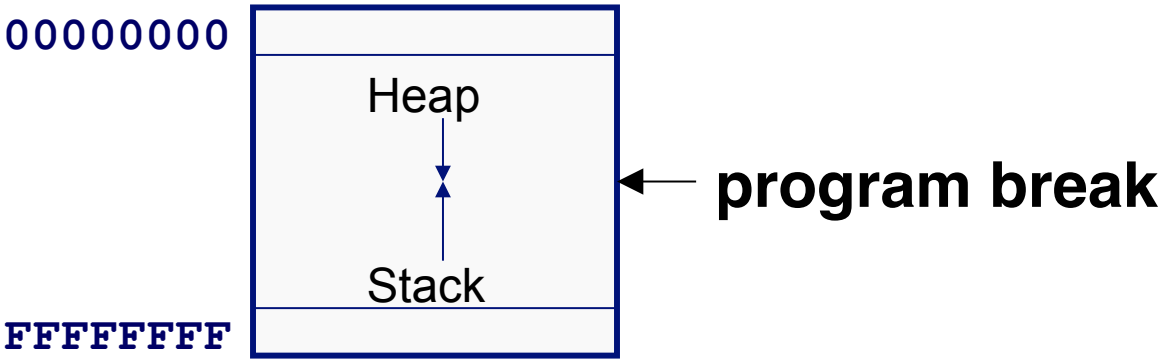


Part 2:

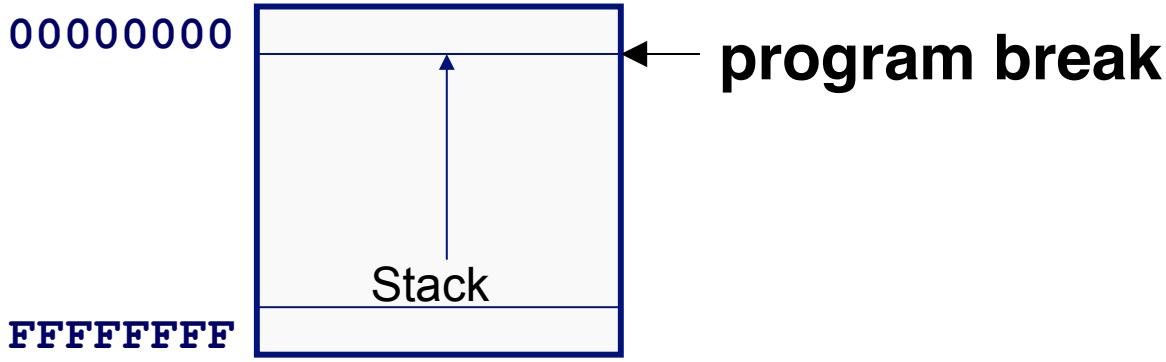
How do **`malloc()`** and **`free()`** work?

The Program Break

The **program break** marks the boundary between heap and stack



Initially, stack has maximum size



Acquiring Heap Memory

Q: How does **`malloc()`** acquire heap memory?

A: Moves program break downward via **`sbrk()`** or **`brk()`** system call

```
void *sbrk(intptr_t increment);
```

- Increment the program break by the specified amount. Calling the function with an increment of 0 returns the current location of the program break. Return 0 if successful and -1 otherwise.
- **Beware: On Linux contains a known bug; should call only with argument 0.**

```
int brk(void *newBreak);
```

- Move the program break to the specified address. Return 0 if successful and -1 otherwise.

Using Heap Memory

Q: Having acquired heap memory, how do **`malloc()`** and **`free()`** manipulate it?

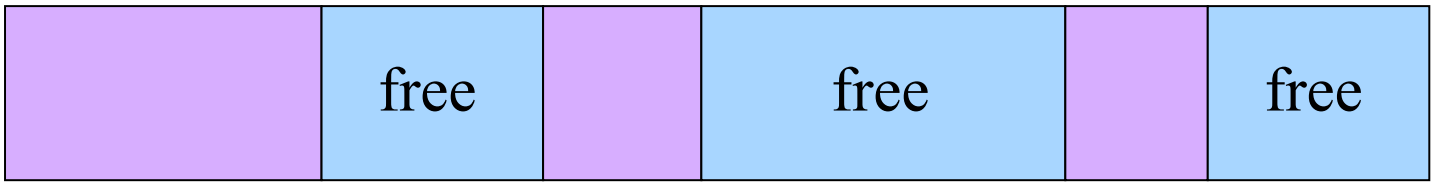
A: Topic of much research; an introduction…

Goals for `malloc()` and `free()`

- Maximizing throughput
  - Maximize number of requests completed per unit time
  - Need both **`malloc()`** and **`free()`** to be fast
- Maximizing memory utilization
  - Minimize the amount of wasted memory
  - Need to minimize size of data structures
- Strawman #1: **`free()`** does nothing
  - Good throughput, but poor memory utilization
- Strawman #2: **`malloc()`** finds the "best fit"
  - Good memory utilization, but poor throughput

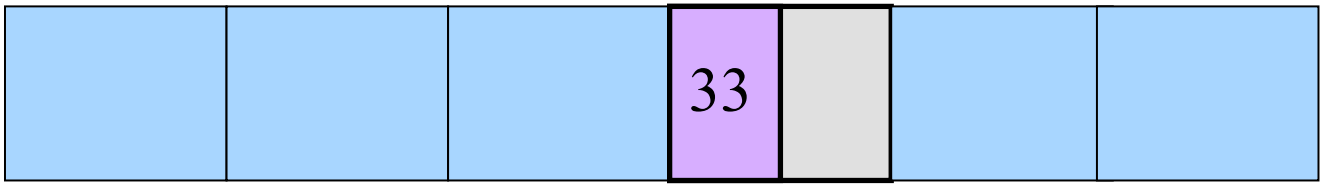
Keeping Track of Free Blocks

- Maintain a list of free blocks of memory
  - Allocate memory from one of the blocks in the free list
  - Deallocate memory by returning the block to the free list
  - When necessary, call **`brk()`** to ask OS for additional memory, and create a new large block
- Design questions
  - How to keep track of the free blocks in memory?
  - How to choose an appropriate free block to allocate?
  - What to do with the left-over space in a free block?
  - What to do with a block that has just been freed?

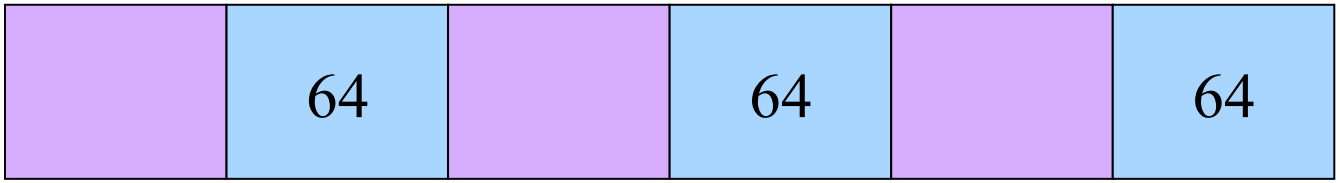


Need to Minimize Fragmentation

- Internal fragmentation
  - Allocated block is larger than **`malloc()`** requested
  - E.g., **`malloc()`** imposes a minimum size (e.g., 64 bytes)



- External fragmentation
  - Enough free memory exists, but no block is big enough
  - E.g., **`malloc()`** asks for 128 contiguous bytes

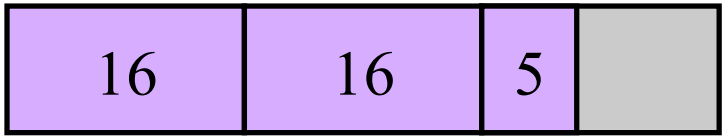


Simple "K&R-Like" Approach

- Memory allocated in multiples of a base size
  - E.g., 16 bytes, 32 bytes, 48 bytes, …
- Linked list of free blocks
  - `malloc()` and `free()` walk through the list to allocate and deallocate
- `malloc()` allocates the *first* big-enough block
  - To avoid sequencing further through the list
- `malloc()` *splits* the free block
  - To allocate what is needed, and leave the rest available
- Linked list is *circular*
  - To be able to continue where you left off
- Linked list in the order the blocks appear in memory
  - To be able to "coalesce" neighboring free blocks

Allocate Memory in Multiples of Base Size

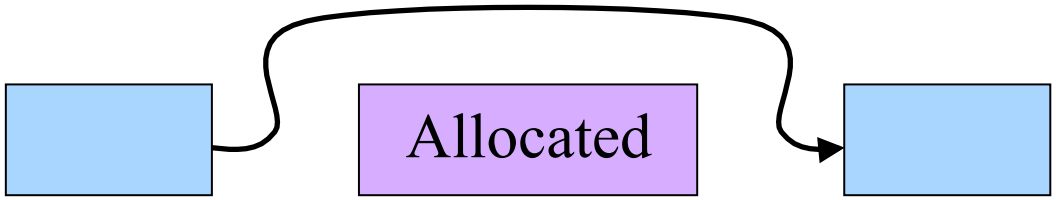
- Allocate memory in multiples of a base size
  - Avoid maintaining very tiny free blocks
  - Align memory on size of largest data type (e.g., double)
- Requested size is "rounded up"
  - Allocation in units of base_size
  - Round: `(nbytes + base_size – 1)/base_size`
- Example:
  - Suppose nbytes is 37
  - And base_size is 16 bytes
  - Then (37 + 16 – 1)/16 is 52/16 which rounds down to 3



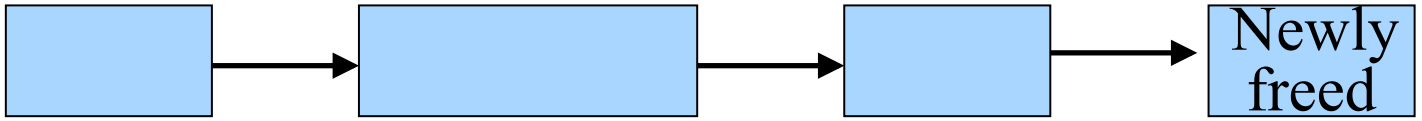
Linked List of Free Blocks

- Linked list of free blocks

- `malloc()` allocates a big-enough block

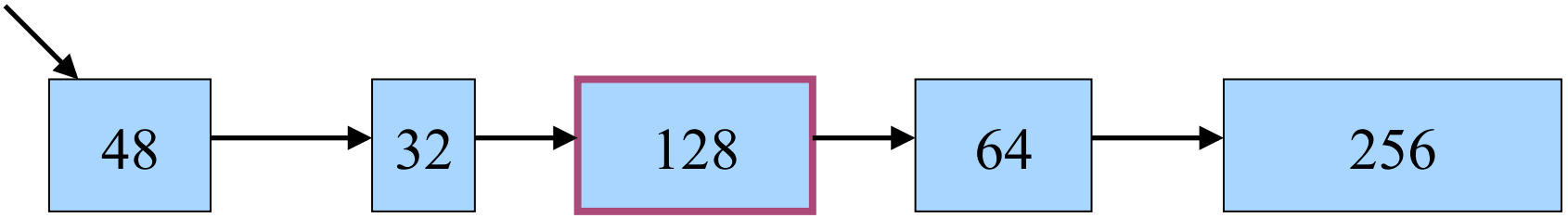


- `free()` adds newly-freed block to the list



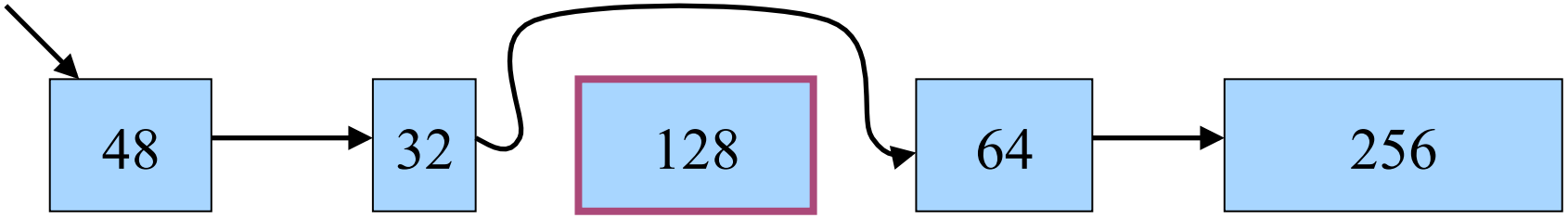
"First-Fit" Allocation

- Handling a request for memory (e.g., **`malloc()`**)
  - Find a free block that satisfies the request
  - Must have a "size" that is big enough, or bigger
- Simplest approach: first fit
  - Sequence through the linked list
  - Stop upon encountering a "big enough" free block
- Example: request for 64 bytes
  - First-fit algorithm stops at the 128-byte block

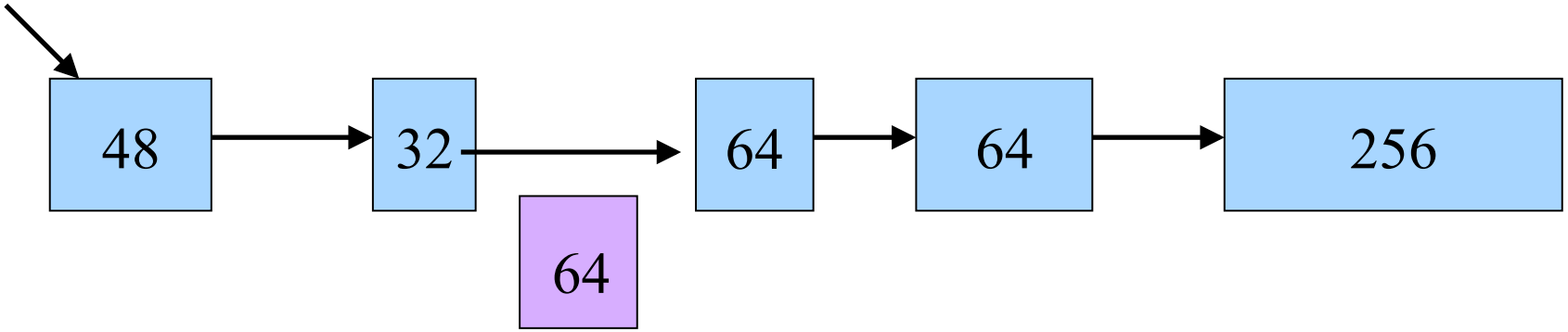


Splitting an Oversized Free Block

- Simple case: perfect fit
  - `malloc()` asks for 128 bytes, free block has 128 bytes
  - Simply remove the free block from the list

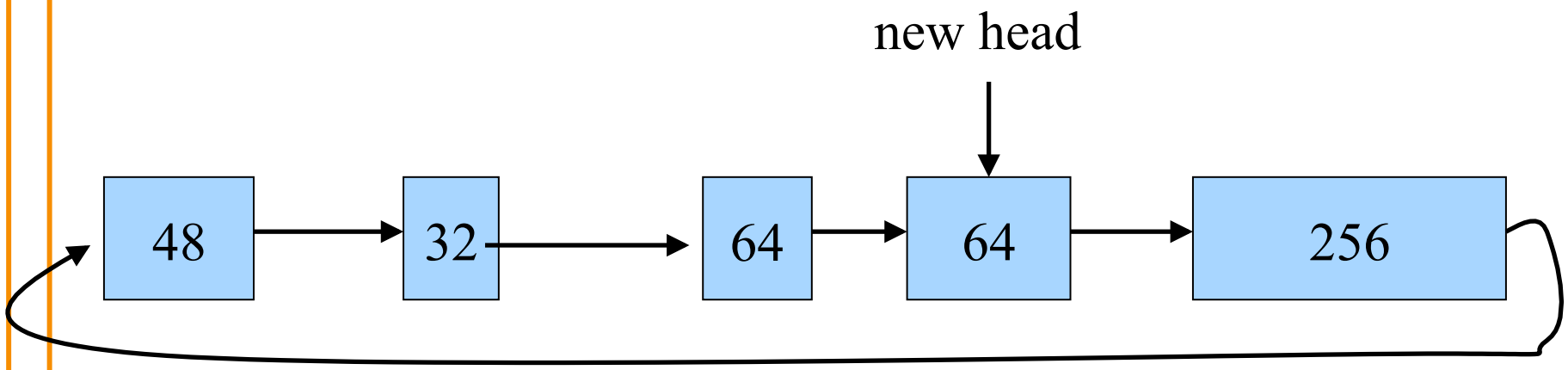


- Complex case: splitting the block
  - `malloc()` asks for 64 bytes, free block has 128 bytes



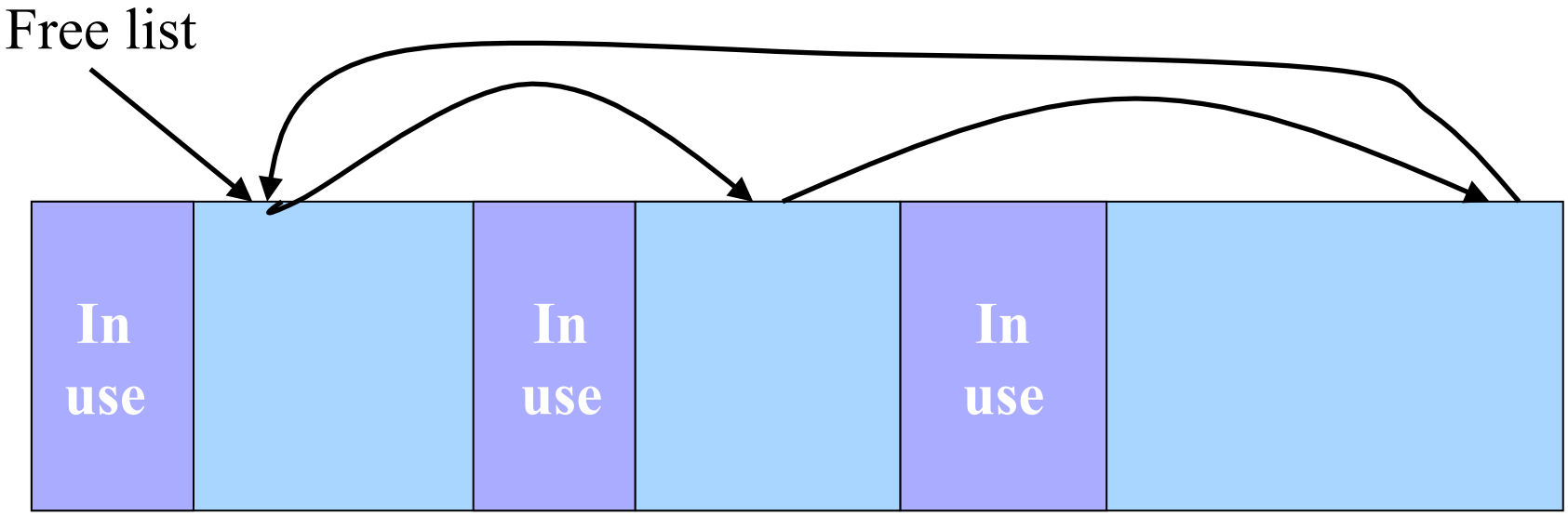
Circular Linked List of Free Blocks

- Advantages of making free list a circular list
  - Any element in the list can be the beginning
  - Don't have to handle the "end" of the list as special
- Performance optimization
  - Make the head be where last block was found
  - More likely to find "big enough" blocks later in the list



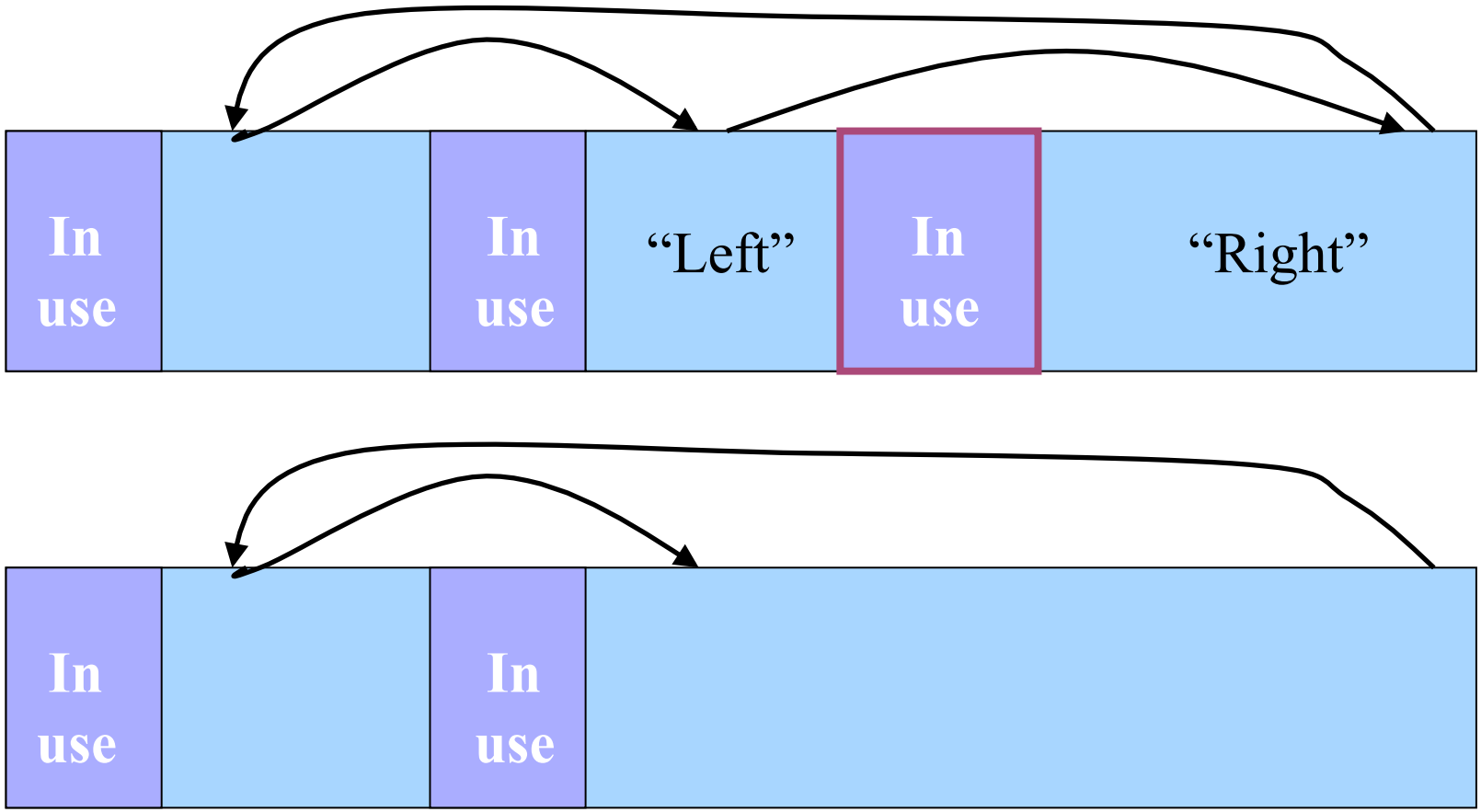
Maintaining Free Blocks in Order

- Keep list in order of increasing addresses
  - Makes it easier to coalesce adjacent free blocks
- Though, makes calls to free() more expensive
  - Need to insert the newly-freed block in the right place



Coalescing Adjacent Free Blocks

- When inserting a block in the free list
  - "Look left" and "look right" for neighboring free blocks



Conclusion

- Elegant simplicity of K&R **`malloc()`** and **`free()`**
  - Simple header with pointer and size in each free block
  - Simple circular linked list of free blocks
  - Relatively small amount of code (~25 lines each)
- Limitations of K&R functions in terms of efficiency
  - **`malloc()`** requires scanning the free list
    - To find the first free block that is big enough
  - **`free()`** requires scanning the free list
    - To find the location to insert the to-be-freed block

Optimizing Dynamic Memory Management

Jennifer Rexford

Goals of this Lecture

- Details of K&R heap manager
- Heap mgr optimizations related to Assignment #6
  - Faster **`free()`** via doubly-linked list, redundant sizes, and status bits
  - Faster **`malloc()`** via binning
- Other heap mgr optimizations
  - Best/good fit block selection
  - Selective splitting
  - Deferred coalescing
  - Segregated data
  - Segregated meta-data
  - Memory mapping

Part 1:

Details of the K&R Heap Manager

An Implementation Challenge

- Need information about each free block
  - Starting address of the block of memory
  - Length of the free block
  - Pointer to the next block in the free list
- Where should this information be stored?
  - Number of free blocks is not known in advance
  - So, need to store the information on the *heap*
- But, wait, this code is what *manages* the heap!!!
  - Can't call **`malloc()`** to allocate storage for this info
  - Can't call **`free()`** to deallocate the storage, either

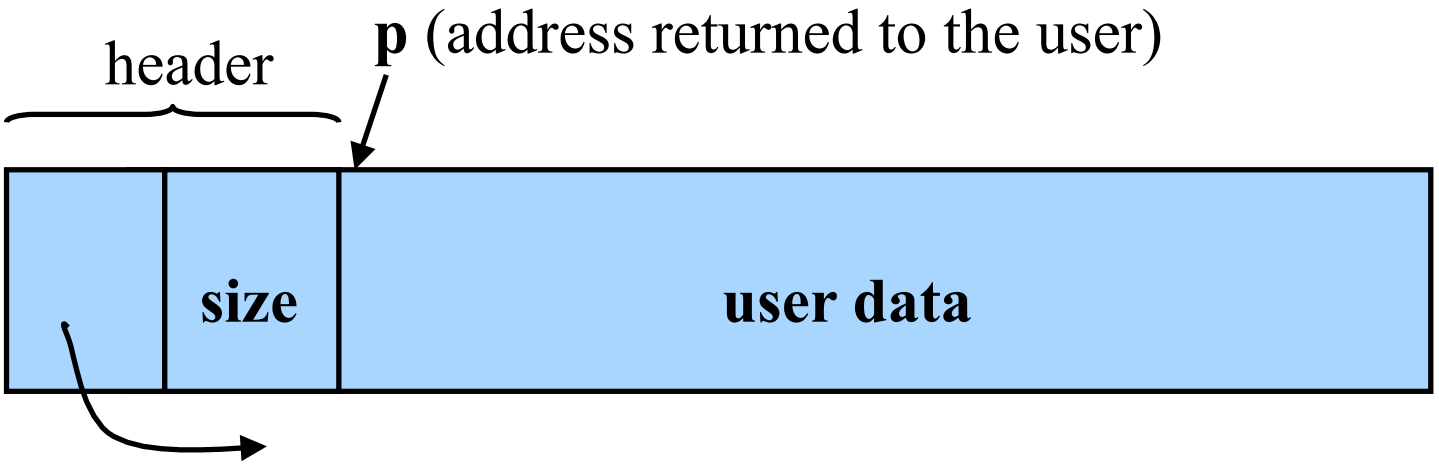
Store Information in the Free Block

Solution:

- Store the information directly in the block
  - Since the memory isn't being used for anything anyway
  - And allows data structure to grow and shrink as needed

Block Headers

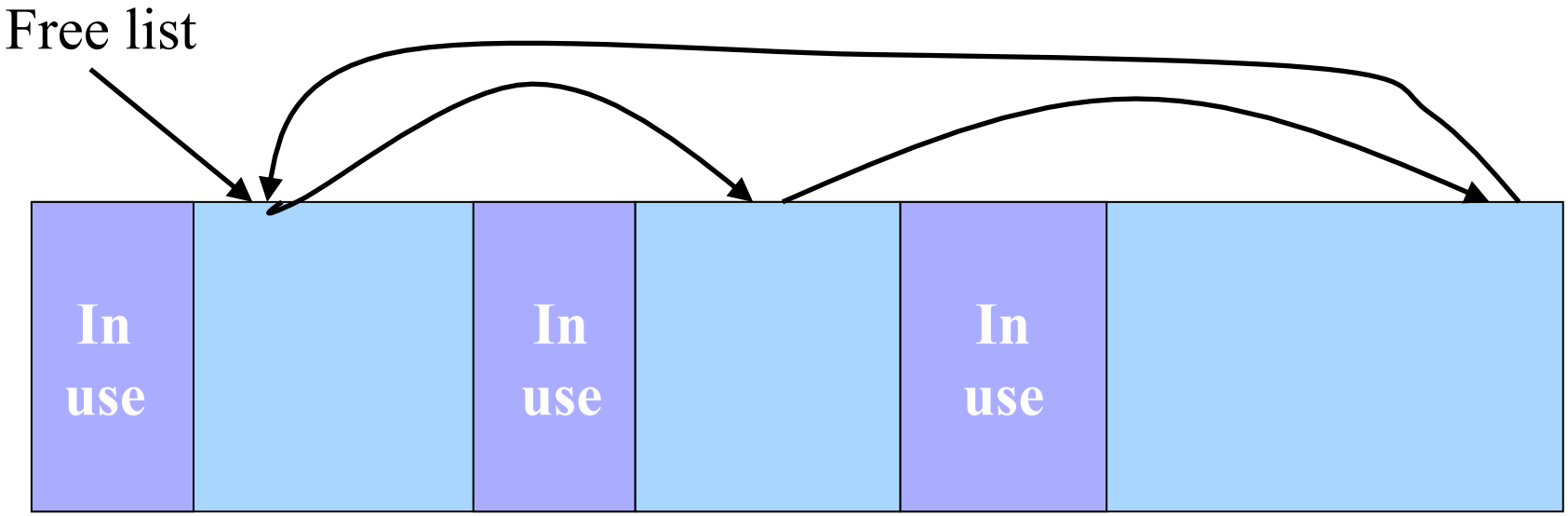
- Every free block has a **header**, containing:
  - Pointer to (i.e., address of) the next free block
  - Size of the free block



- Challenge: programming outside the type system

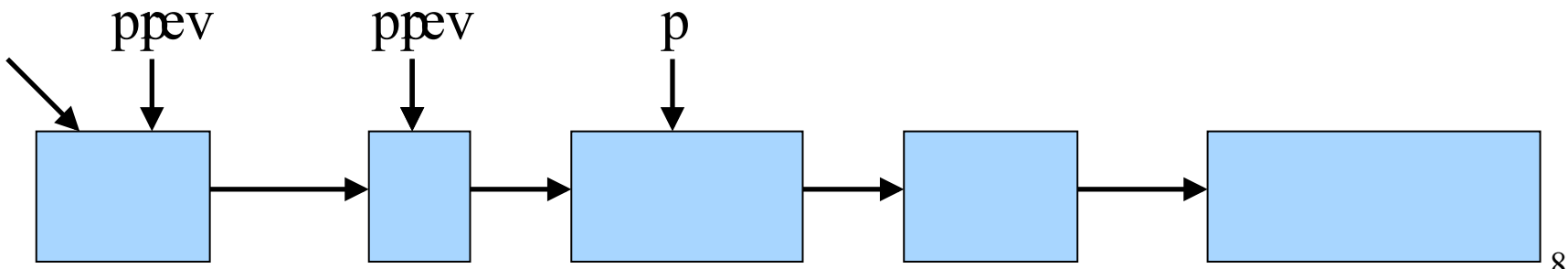
Free List: Circular Linked List

- Free blocks, linked together
  - Example: circular linked list
- Keep list in order of increasing addresses
  - Makes it easier to coalesce adjacent free blocks



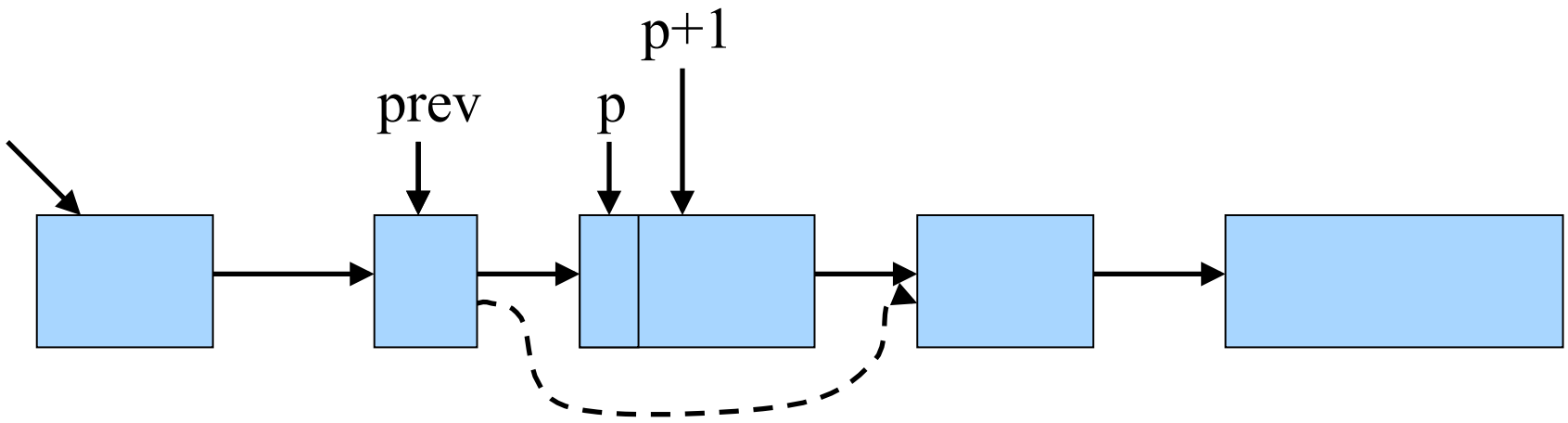
Malloc: First-Fit Algorithm

- Start at the beginning of the list
- Sequence through the list
  - Keep a pointer to the previous element
- Stop when reaching first block that is big enough
  - Patch up the list
  - Return a pointer to the user



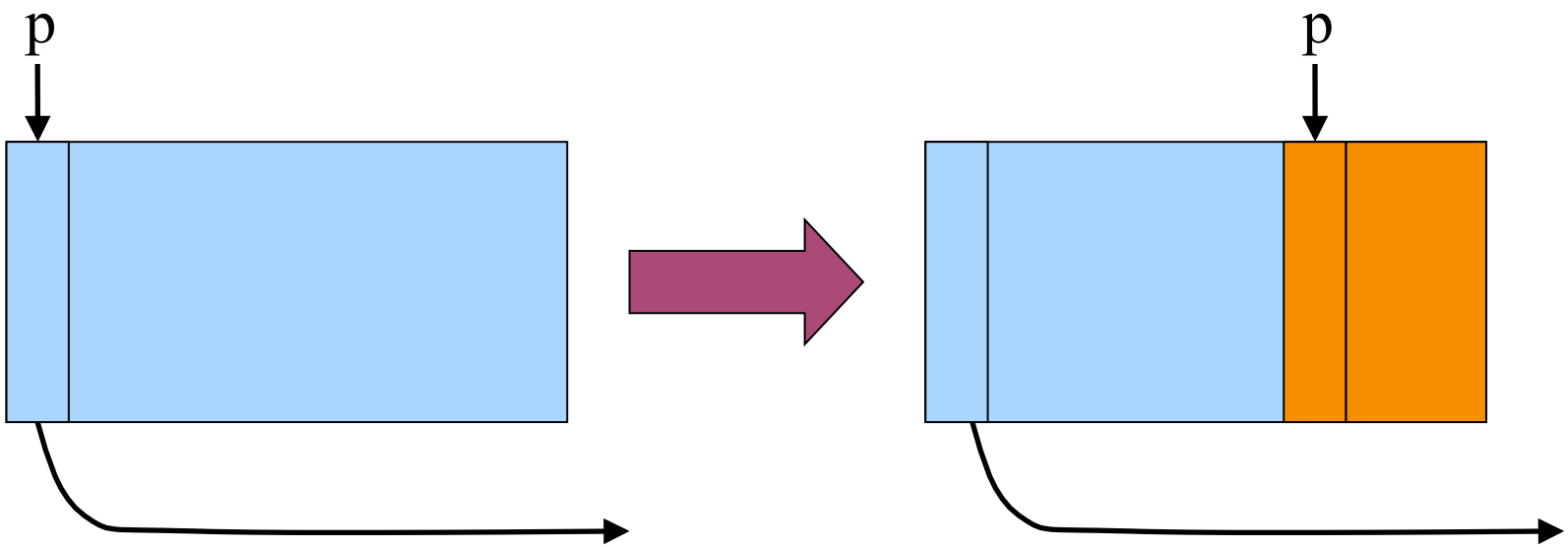
Malloc: First Case: Perfect Fit

- Suppose the first fit is a perfect fit
  - Remove the block from the list
  - Link the previous free block with the next free block
  - Return the current to the user (skipping header)



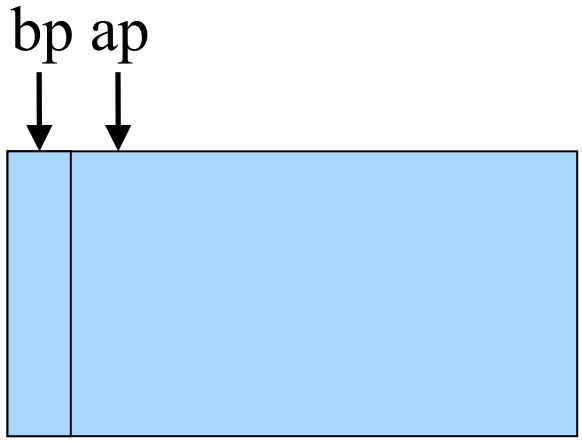
Malloc: Second Case: Big Block

- Suppose the block is bigger than requested
  - Divide the free block into two blocks
  - Keep first (now smaller) block in the free list
  - Allocate the second block to the user
  - Bonus: No need to manipulate links



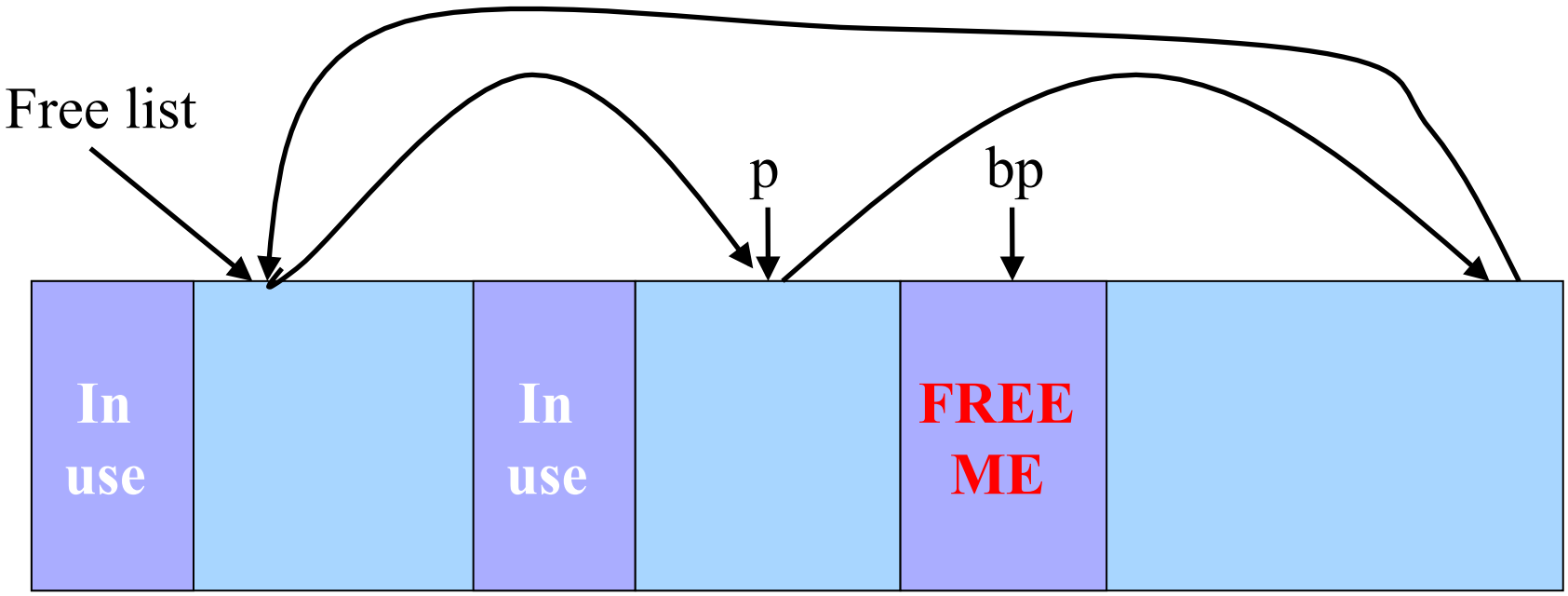
Free

- User passes a pointer to the memory block
  - `void free(void *ap);`
- `free()` function inserts block into the list
  - Identify the start of entry
  - Find the location in the free list
  - Add to the list, coalescing entries, if needed



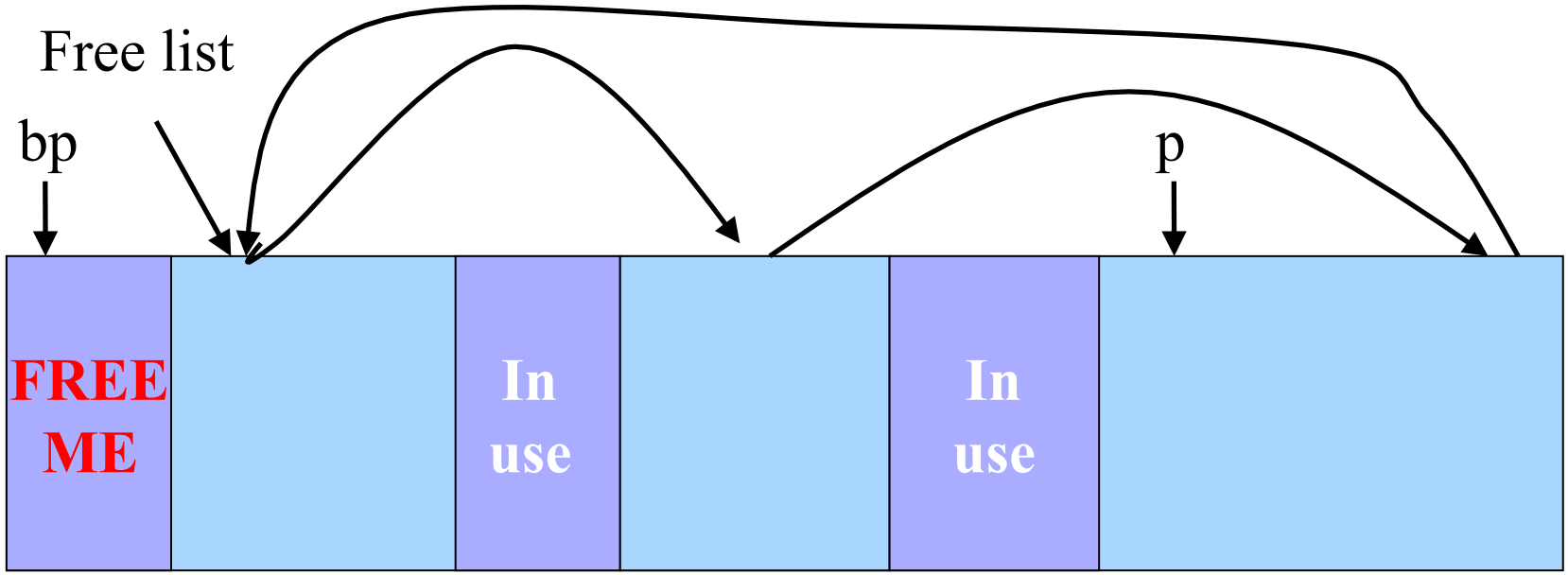
Free: Finding Location to Insert

- Start at the beginning
- Sequence through the list
- Stop at last entry before the to-be-freed element



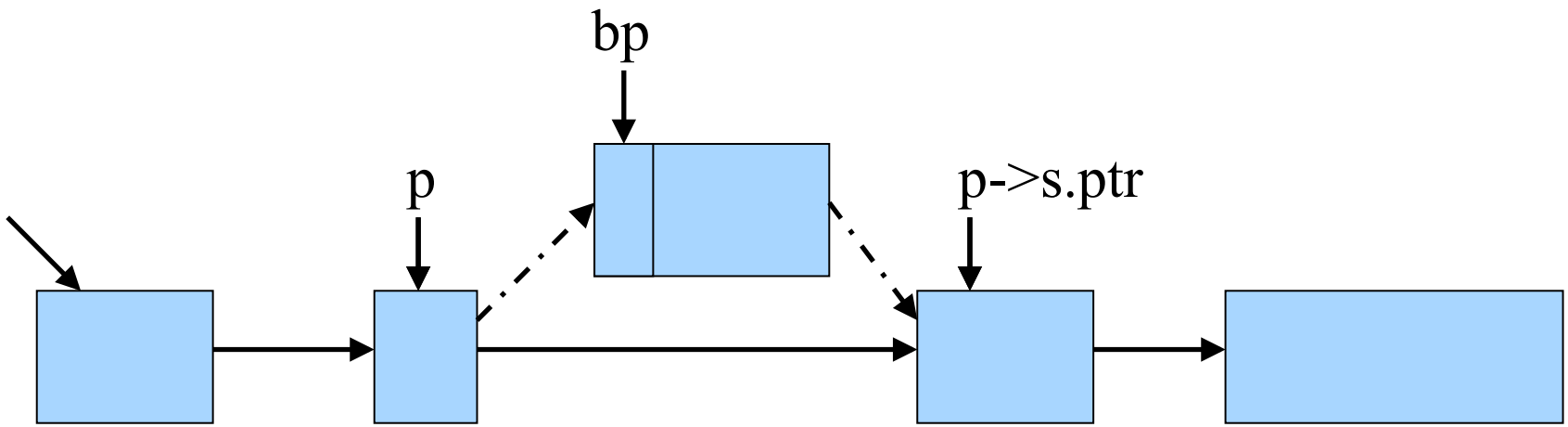
Free: Handling Corner Cases

- Check for wrap-around in memory
  - To-be-freed block is before first entry in the free list, or
  - To-be-freed block is after the last entry in the free list



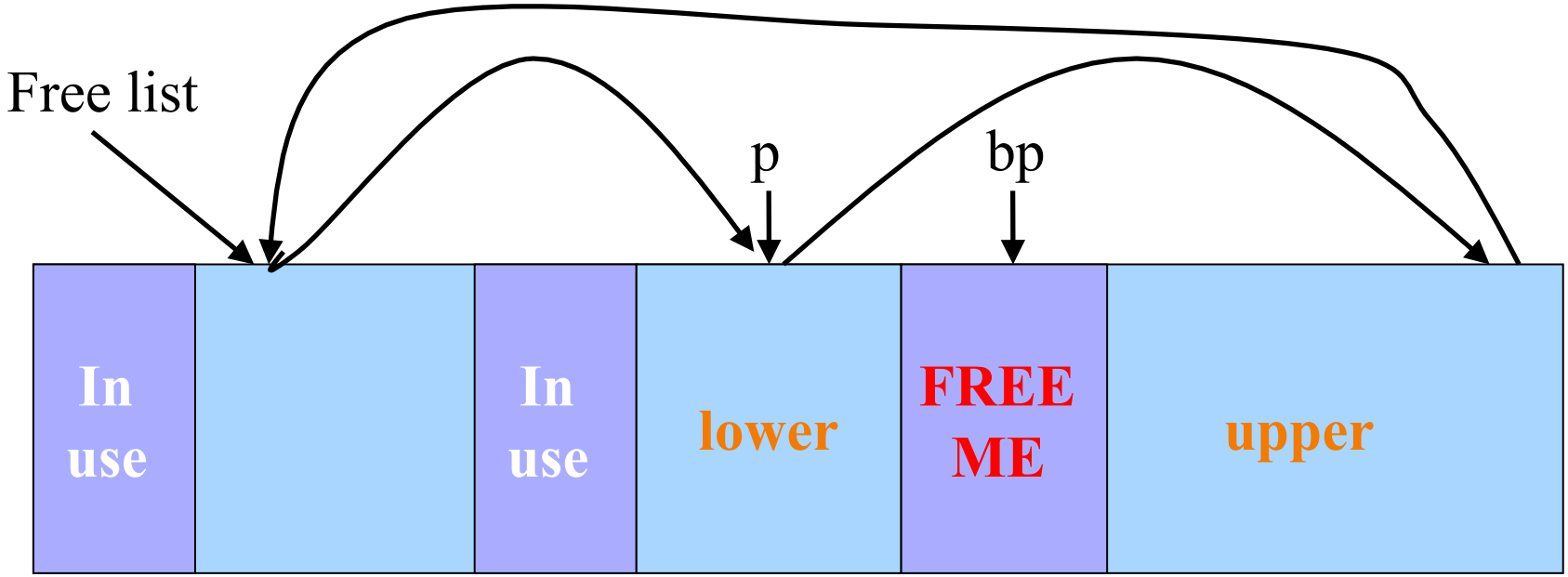
Free: Inserting Into Free List

- New element to add to free list
- Insert in between previous and next entries
- But, there may be opportunities to coalesce



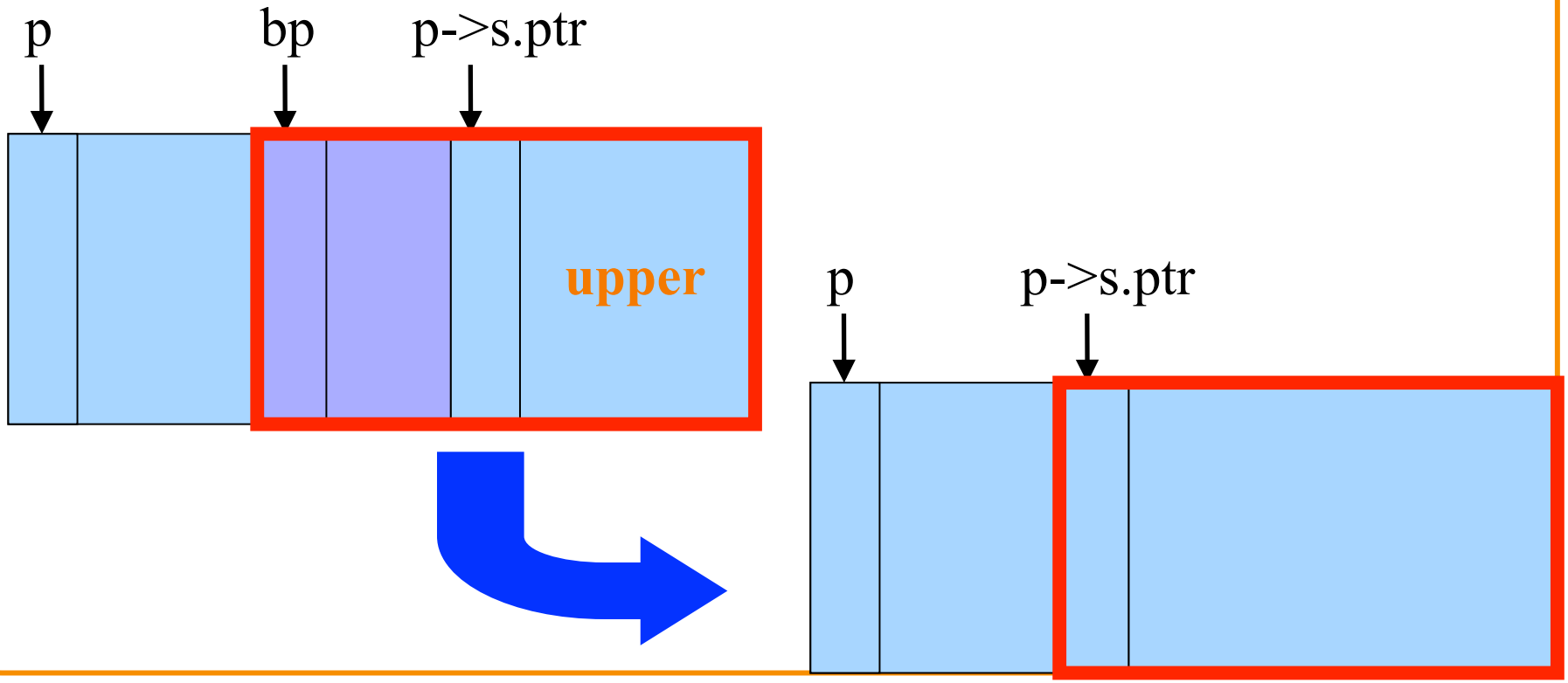
Coalescing With Neighbors

- Scanning the list finds the location for inserting
  - Pointer to to-be-freed element: **`bp`**
  - Pointer to previous element in free list: **`p`**
- Coalescing into larger free blocks
  - Check if contiguous to upper and lower neighbors



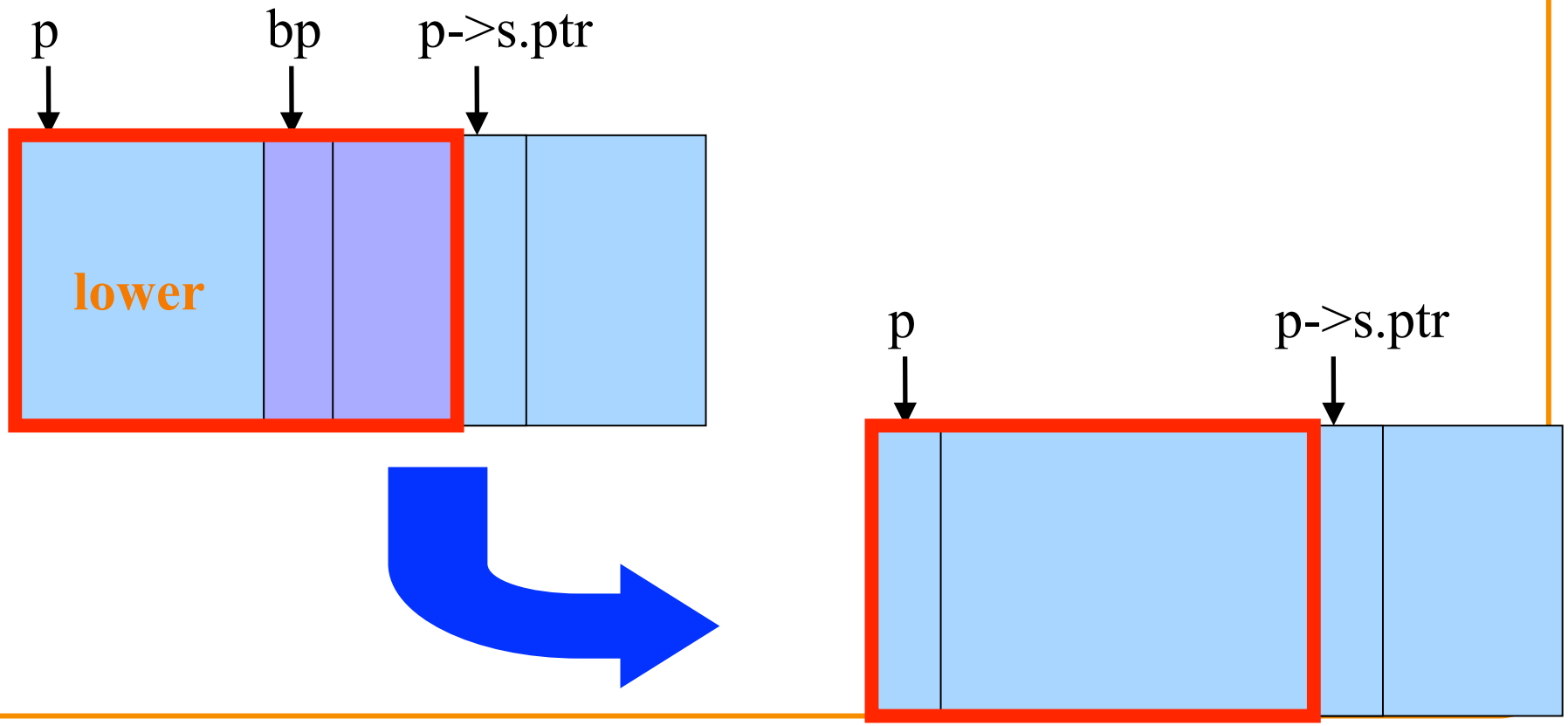
Coalesce With Upper Neighbor

- Check if next part of memory is in the free list
- If so, make into one bigger block
- Else, simply point to the next free element



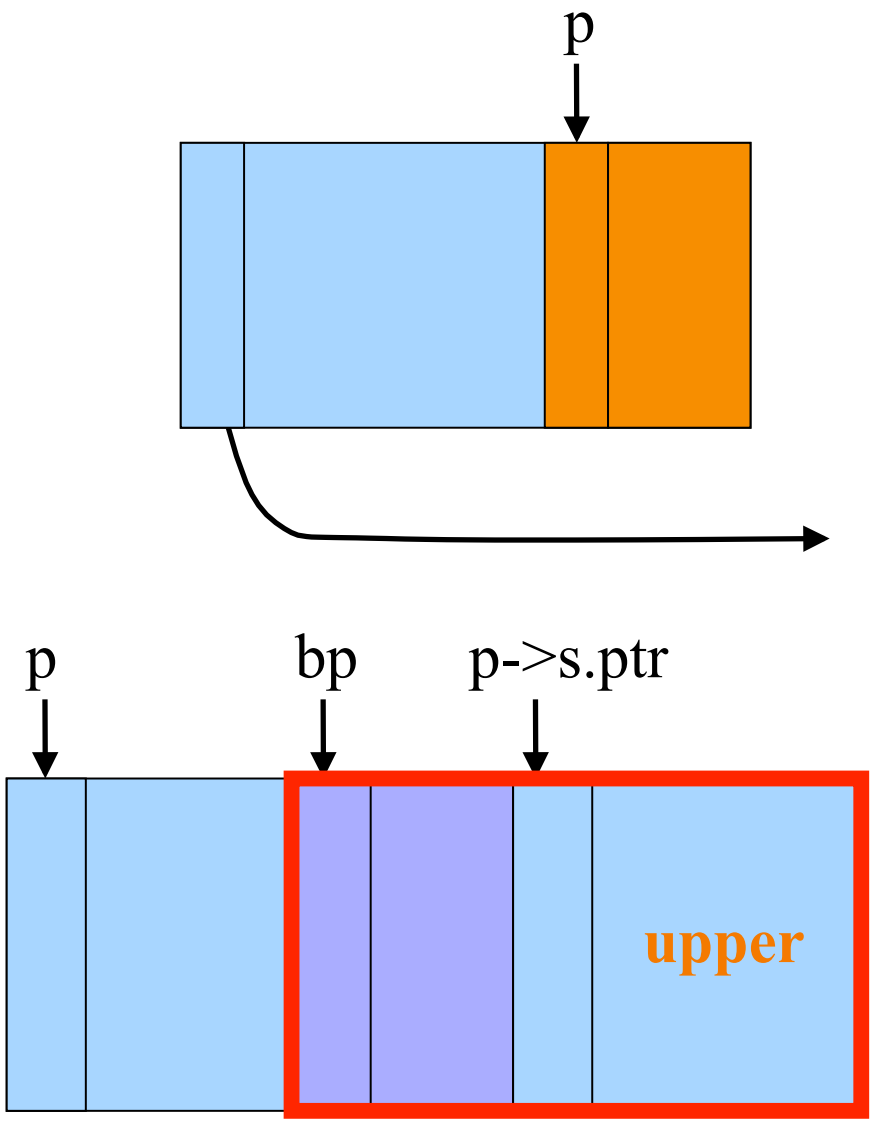
Coalesce With Lower Neighbor

- Check if previous part of memory is in the free list
- If so, make into one bigger block



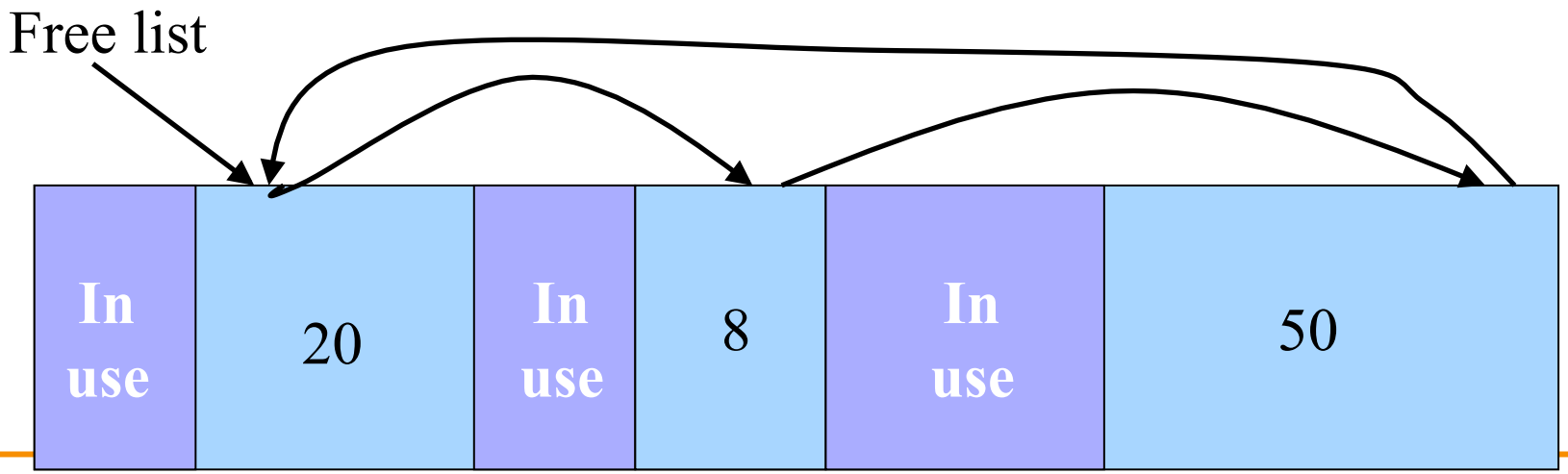
Strengths of K&R Approach

- Advantages
  - Simplicity of the code
- Optimizations to `malloc()`
  - Splitting large free block to avoid wasting space
- Optimization to `free()`
  - Roving free-list pointer is left at the last place a block was allocated
  - Coalescing contiguous free blocks to reduce fragmentation



Weaknesses of K&R Approach

- Inefficient use of memory: fragmentation
  - First-fit policy can leave lots of "holes" of free blocks in memory
- Long execution times: linear-time overhead
  - `malloc()` scans the free list to find a big-enough block
  - `free()` scans the free list to find where to insert a block
- Accessing a wide range of memory addresses in free list
  - Can lead to large amount of paging to/from the disk

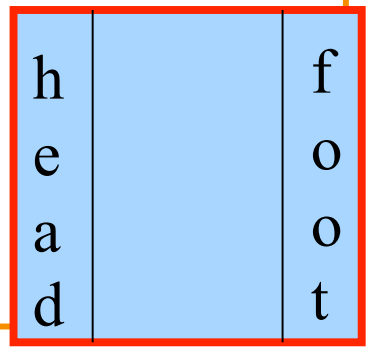


Part 2:

Optimizations Related to Assignment 6

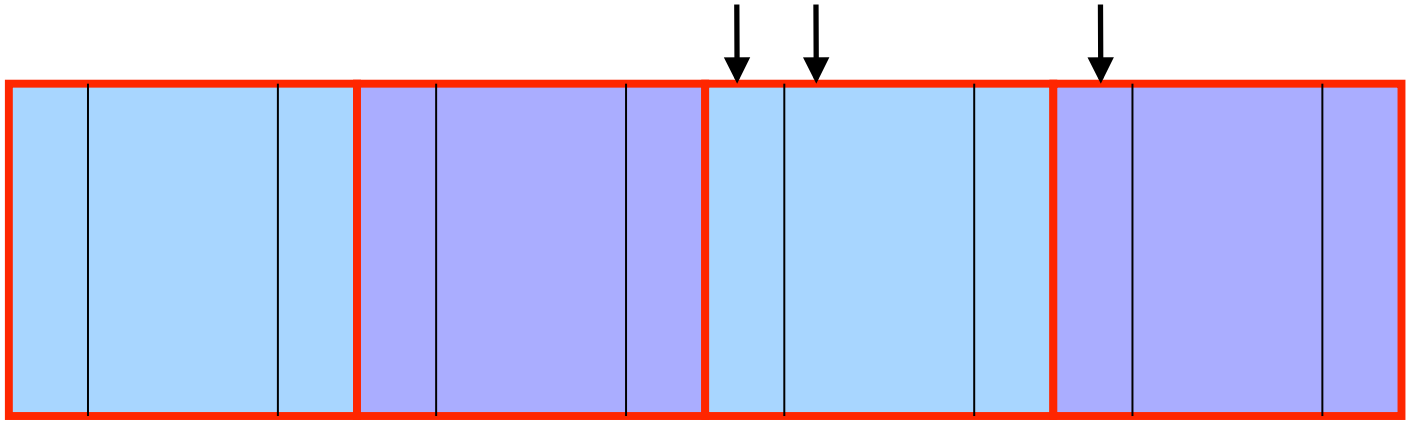
Faster Free

- Performance problems with K&R `free()`
  - Scanning the free list to know where to insert
  - Keeping track of the "previous" node to do the insertion
- Doubly-linked, non-circular list
  - Header
    - Size of the block (in # of units)
    - Flag indicating whether the block is free or in use
    - If free, a pointer to the next free block
  - Footer
    - Size of the block (in # of units)
    - If free, a pointer to the previous free block



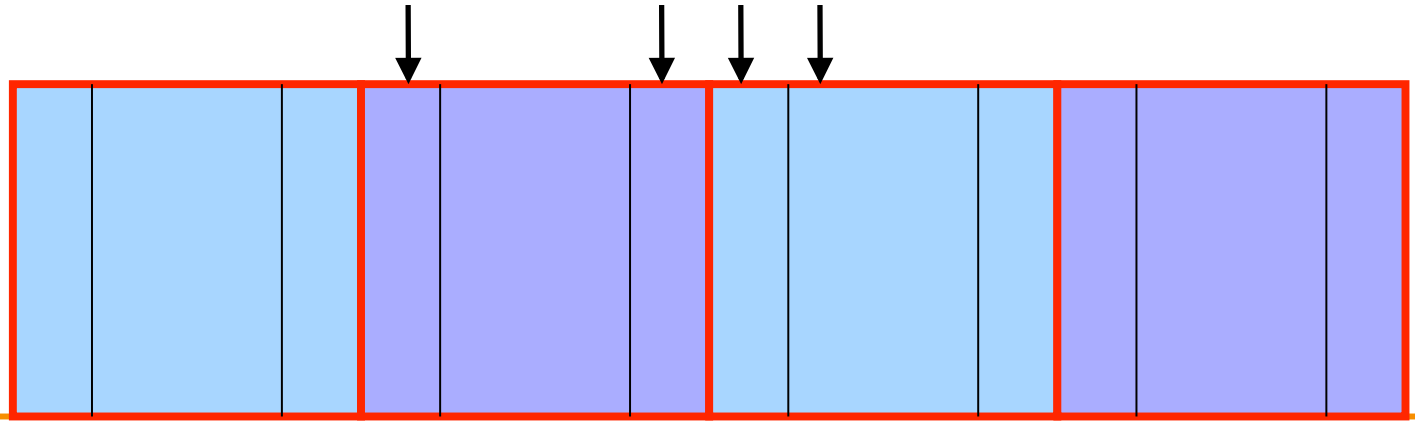
Size: Finding Next Block

- Go quickly to next block in memory
  - Start with the user's data portion of the block
  - Go backwards to the head of the block
    - Easy, since you know the size of the header
  - Go forward to the head of the next block
    - Easy, since you know the size of the current block



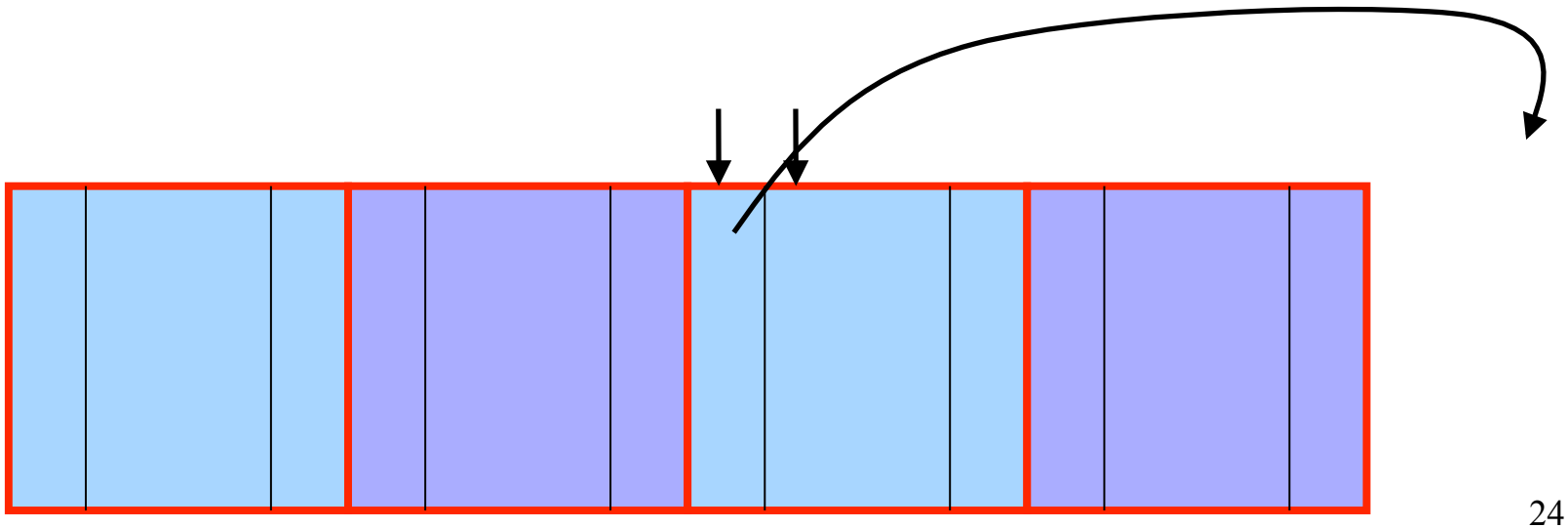
Size: Finding Previous Block

- Go quickly to previous chunk in memory
  - Start with the user's data portion of the block
  - Go backwards to the head of the block
    - Easy, since you know the size of the header
  - Go backwards to the footer of the previous block
    - Easy, since you know the size of the footer
  - Go backwards to the header of the previous block
    - Easy, since you know the size from the footer



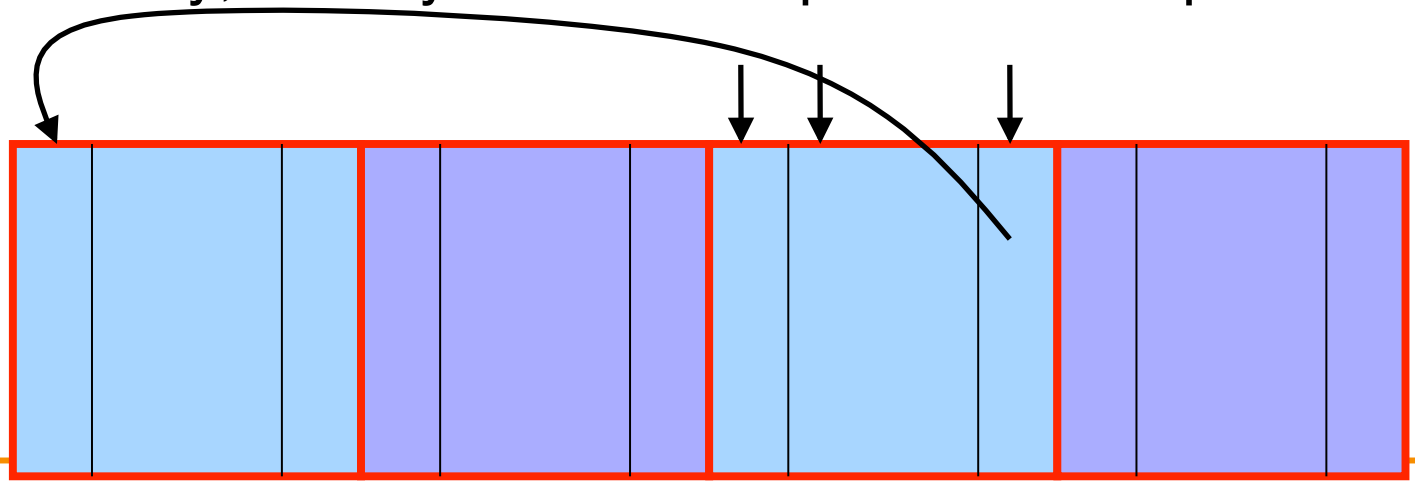
Pointers: Next Free Block

- Go quickly to next free block in memory
  - Start with the user's data portion of the block
  - Go backwards to the head of the block
    - Easy, since you know the size of the header
  - Go forwards to the next free block
    - Easy, since you have the next free pointer



Pointers: Previous Free Block

- Go quickly to previous free block in memory
  - Start with the user's data portion of the block
  - Go backwards to the head of the block
    - Easy, since you know the size of the header
  - Go forwards to the footer of the block
    - Easy, since you know the block size from the header
  - Go backwards to the previous free block
    - Easy, since you have the previous free pointer

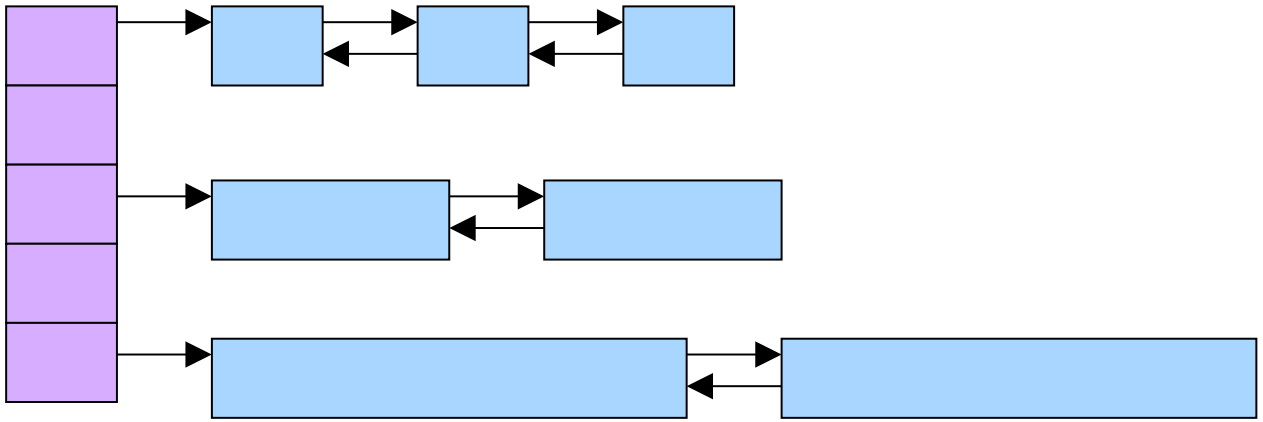


Efficient Free

- Before: K&R
  - Scan the free list till you find the place to insert
    - Needed to see if you can coalesce adjacent blocks
  - Expensive for loop with several pointer comparisons
- After: with header/footer and doubly-linked list
  - Coalescing with the previous block in memory
    - Check if previous block in memory is also free
    - If so, coalesce
  - Coalescing with the next block in memory the same way
  - Add the new, larger block to the front of the linked list

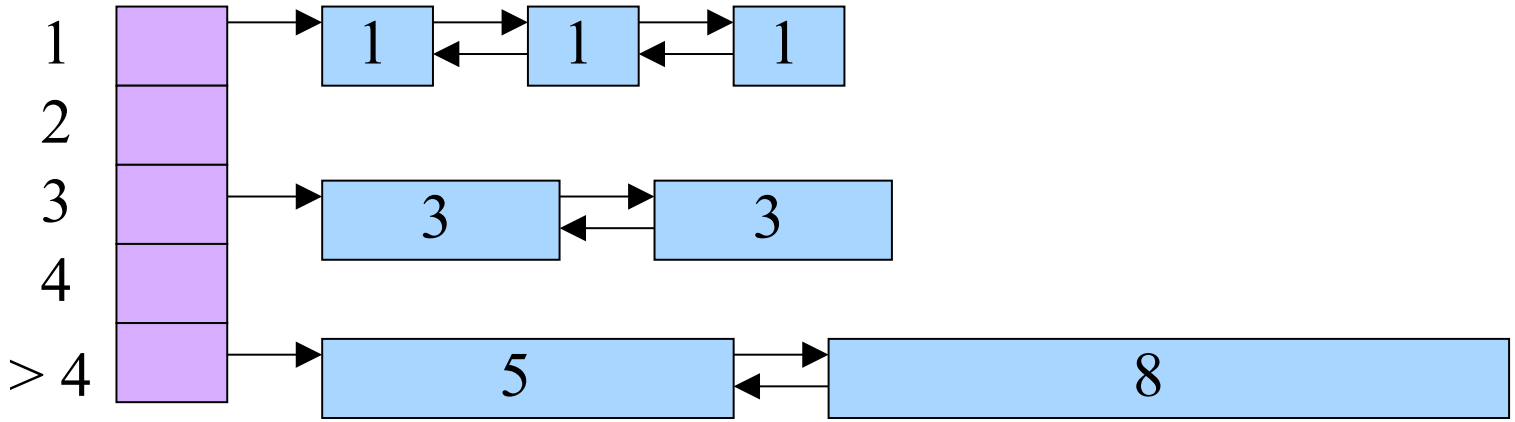
But Malloc is Still Slow…

- Still need to scan the free list
  - To find the first, or best, block that fits
- Root of the problem
  - Free blocks have a wide range of sizes
- Solution: binning
  - Separate free lists by block size
  - Implemented as an array of free-list pointers



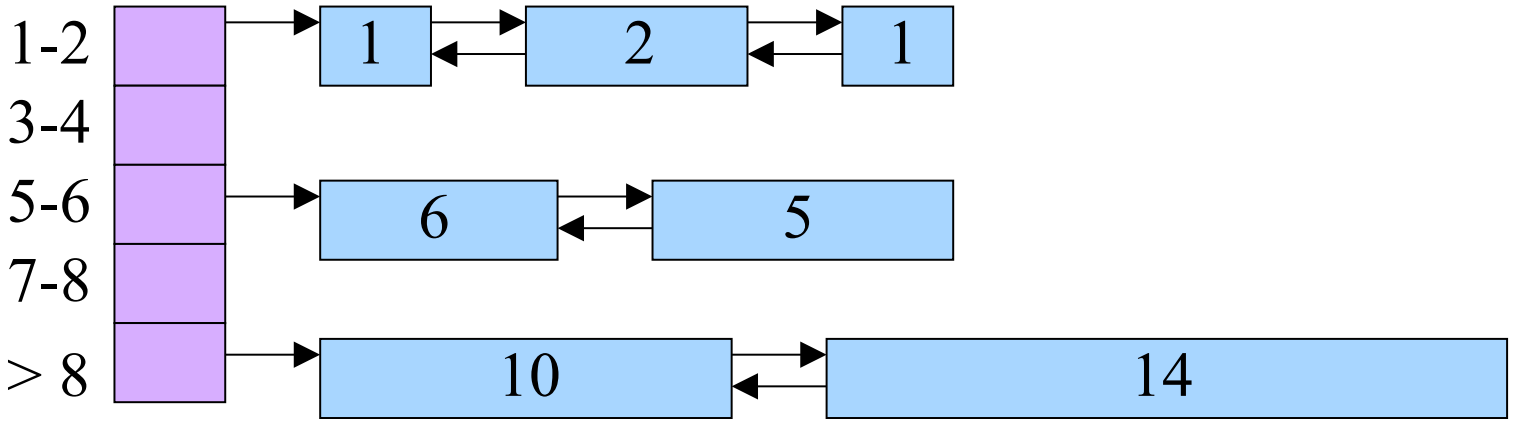
Binning Strategies: Exact Fit

- Have a bin for each block size, up to a limit
  - Advantages: no search for requests up to that size
  - Disadvantages: many bins, each storing a pointer
- Except for a final bin for all larger free blocks
  - For allocating larger amounts of memory
  - For splitting to create smaller blocks, when needed



Binning Strategies: Range

- Have a bin cover a range of sizes, up to a limit
  - Advantages: fewer bins
  - Disadvantages: need to search for a big enough block
- Except for a final bin for all larger free chunks
  - For allocating larger amounts of memory
  - For splitting to create smaller blocks, when needed



Suggestions for Assignment #6

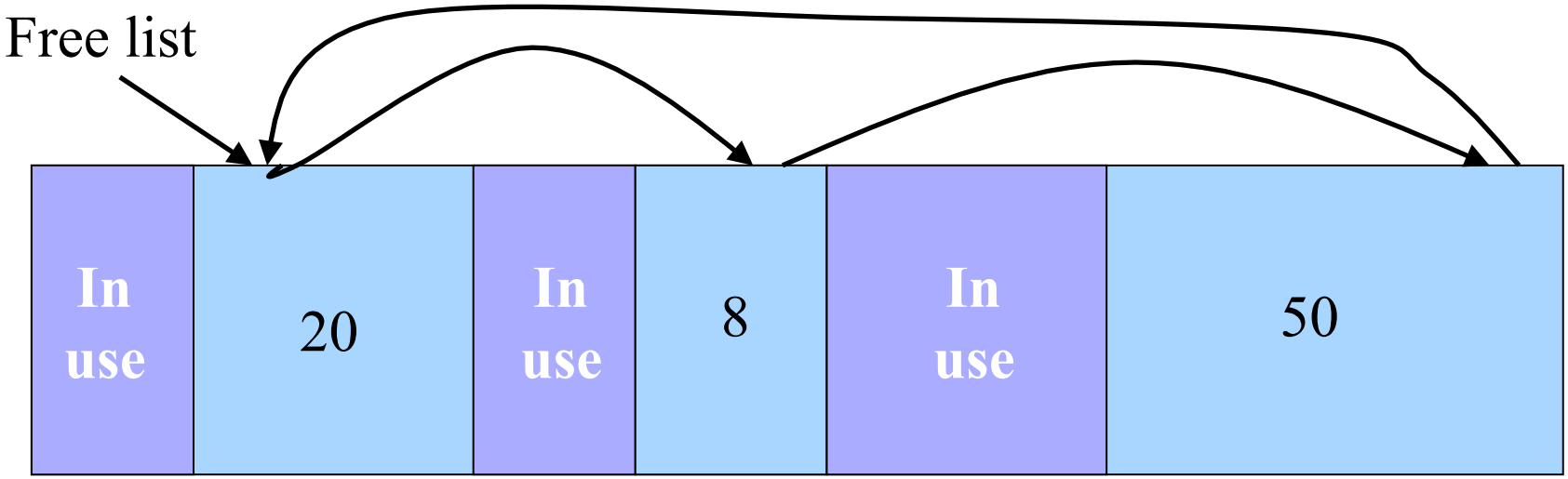
- Debugging memory management code is hard
  - A bug in your code might stomp on the headers or footers
  - … making it very hard to understand where you are in memory
- Suggestion: debug carefully as you go along
  - Write little bits of code at a time, and test as you go
  - Use assertion checks very liberally to catch mistakes early
  - Use functions to apply higher-level checks on your list
    - E.g,. all free-list blocks are marked as free
    - E.g., each block pointer is within the heap range
    - E.g., the block size in header and footer are the same
- Suggestion: draw lots and lots of pictures

Part 3:

Other Optimizations

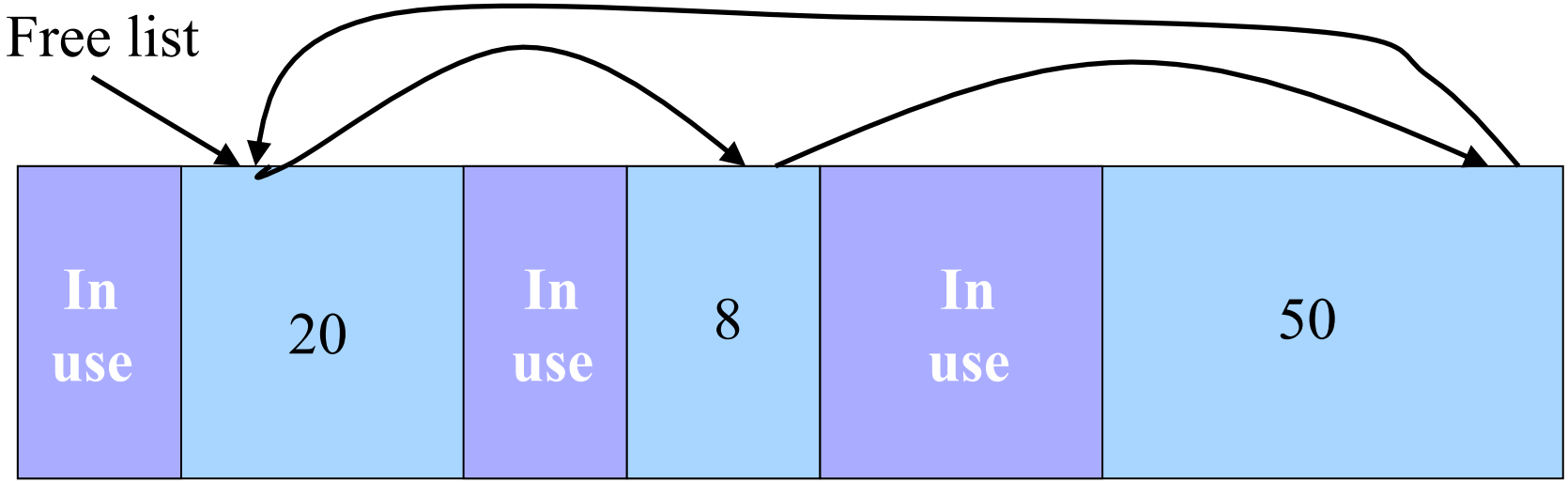
Best/Good Fit Block Selection

- Observation:
  - K&R uses "first fit" (really, "next fit") strategy
  - Example: `malloc(8)` would choose the 20-byte block
- Alternative: "**best fit**" or "**good fit**" strategy
  - Example: `malloc(8)` would choose the 8-byte block
  - Applicable if not binning, or if a bin has blocks of variable sizes
  - **Pro**: Minimizes internal fragmentation and splitting
  - **Con**: Increases cost of choosing free block



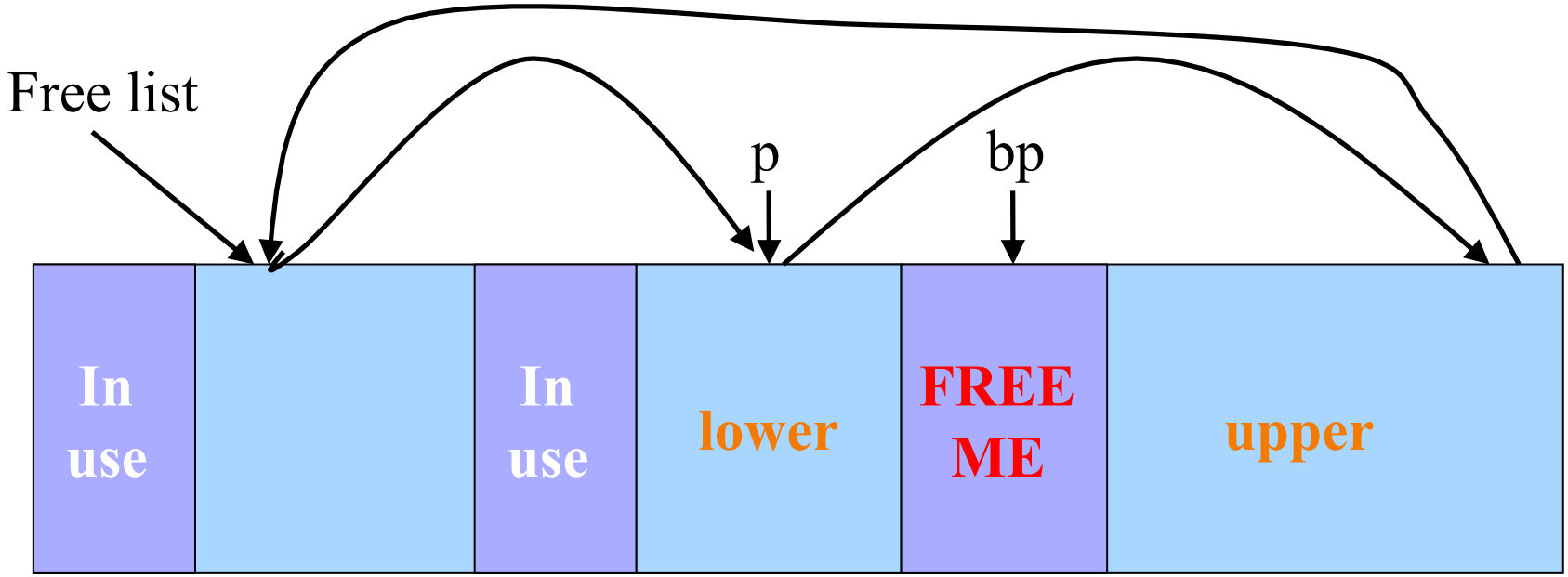
Selective Splitting

- Observation:
  - K&R `malloc()` splits whenever chosen block is too big
  - Example: `malloc(14)` splits the 20-byte block
- Alternative: **selective splitting**
  - Split only when the saving is big enough
  - Example: `malloc(14)` allocates the entire 20-byte block
  - **Pro**: Reduces external fragmentation
  - **Con**: Increases internal fragmentation



Deferred Coalescing

- Observation:
  - K&R does coalescing in `free()` whenever possible
- Alternative: **deferred coalescing**
  - Wait, and coalesce many blocks at a later time
  - **Pro**: Handles "`malloc(x);free();malloc(x)`" sequences well
  - **Con**: Complicates algorithms



Segregated Data

- Observation:
  - Splitting and coalescing consume lots of overhead
- Problem:
  - How to eliminate that overhead?
- Solution: **Segregated data**
  - **Make use of the virtual memory concept…**
  - Store each bin's blocks in a distinct virtual memory page
  - Elaboration…

Segregated Data (cont.)

- Segregated data
  - Each bin contains blocks of fixed sizes
    - E.g. 32, 64, 128, …
  - All blocks within a bin are from same **virtual memory** page
  - Malloc never splits! Examples:
    - Malloc for 32 bytes => provide 32
    - Malloc for 5 bytes => provide 32
    - Malloc for 100 bytes => provide 128
  - Free never coalesces!
    - Free block => examine address, infer virtual memory page, infer bin, insert into that bin
  - **Pro**: Completely eliminates splitting and coalescing overhead
  - **Pro**: Eliminates most meta-data; only forward links are required (no backward links, sizes, status bits, footers)
  - **Con**: Some usage patterns cause excessive external fragmentation

Segregated Meta-Data

- Observations:
  - Meta-data (block sizes, status flags, links, etc.) are scattered across the heap, interspersed with user data
  - Heap mgr often must traverse meta-data
- Problem 1:
  - User error easily can corrupt meta-data
- Problem 2:
  - Frequent traversal of meta-data can cause excessive page faults
- Solution: **Segregated meta-data**
  - **Make use of the virtual memory concept…**
  - Store meta-data in a distinct (segregated) virtual memory page from user data

Memory Mapping

- Observations:
  - Heap mgr might want to release heap memory to OS (e.g. for use as stack)
  - Heap mgr can call **`brk(currentBreak-x)`** to release freed memory to OS, but…
  - Difficult to know when memory at high end of heap is free, and…
  - Often freed memory is not at high end of heap!
- Problem:
  - How can heap mgr effectively release freed memory to OS?
- Solution: **Memory mapping**
  - **Make use of virtual memory concept…**
  - Allocate memory via **`mmap()`** system call
  - Free memory via **`munmap()`** system call

`mmap()` and `munmap()`

- Typical call of **`mmap()`**

```
p = mmap(NULL, size, PROT_READ|PROT_WRITE,
         MAP_PRIVATE|MAP_ANON, 0, 0);
```

  - Asks the OS to map a new private read/write area of virtual memory containing **`size`** bytes
  - Returns the virtual address of the new area on success, NULL on failure

- Typical call of **`munmap()`**

```
status = munmap(p, size);
```

  - Unmaps the area of virtual memory at virtual address **`p`** consisting of **`size`** bytes
  - Returns 1 on success, 0 on failure

- See Bryant & O'Hallaron book and man pages for details

Using `mmap()` and `munmap()`

Typical strategy:

- Allocate **small** block =>
  - Call **`brk()`** if necessary
  - Manipulate data structures described earlier in this lecture
- Free **small** block =>
  - Manipulate data structures described earlier in this lecture
  - Do not call **`brk()`**

- Allocate **large** block =>
  - Call **`mmap()`**
- Free **large** block =>
  - Call **`munmap()`**

Summary

- Details of K&R heap manager
- Heap mgr optimizations related to Assignment #6
  - Faster **`free()`** via doubly-linked list, redundant sizes, and status bits
  - Faster **`malloc()`** via binning
- Other heap mgr optimizations
  - Best/good fit block selection
  - Selective splitting
  - Deferred coalescing
  - Segregated data
  - Segregated meta-data
  - Memory mapping

I/O and Process Management

Jennifer Rexford

Goals of this Lecture: I/O

- The Unix **stream** concept
- Standard C I/O functions
- Unix system-level functions for I/O
- How C I/O functions use the Unix functions
- Additional abstractions provided by C I/O functions

Streams are a beautiful Unix abstraction

Stream Abstraction

- Any source of input or destination for output
  - E.g., keyboard as input, and screen as output
  - E.g., files on disk or CD, network ports, printer port, …
- Accessed in C programs through file pointers
  - E.g., `FILE *fp1, *fp2;`
  - E.g., `fp1 = fopen("myfile.txt", "r");`
- Three streams provided by stdio.h
  - Streams **stdin**, **stdout**, and **stderr**
    - Typically map to keyboard, screen, and screen
  - Can redirect to correspond to other streams
    - E.g., stdin can be the output of another program
    - E.g., stdout can be the input to another program

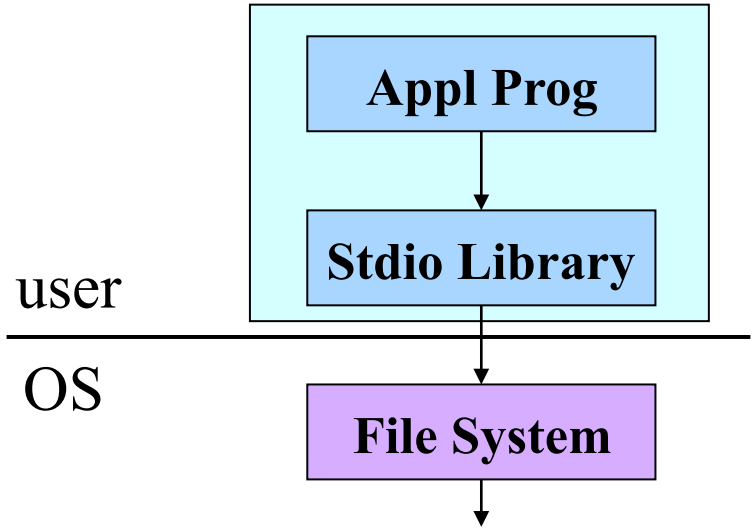
Sequential Access to a Stream

- Each stream has an associated file position
  - Starting at beginning of file (if opened to read or write)
  - Or, starting at end of file (if opened to append)

- Read/write operations advance the file position
  - Allows sequencing through the file in sequential manner
- Support for random access to the stream
  - Functions to learn current position and seek to new one

Standard I/O Functions

- Portability
  - Generic I/O support for C programs
  - Specific implementations for various host OSes
  - Invokes the OS-specific system calls for I/O
- Abstractions for C programs
  - Streams
  - Line-by-line input
  - Formatted output
- Additional optimizations
  - Buffered I/O
  - Safe writing



Example: Opening a File

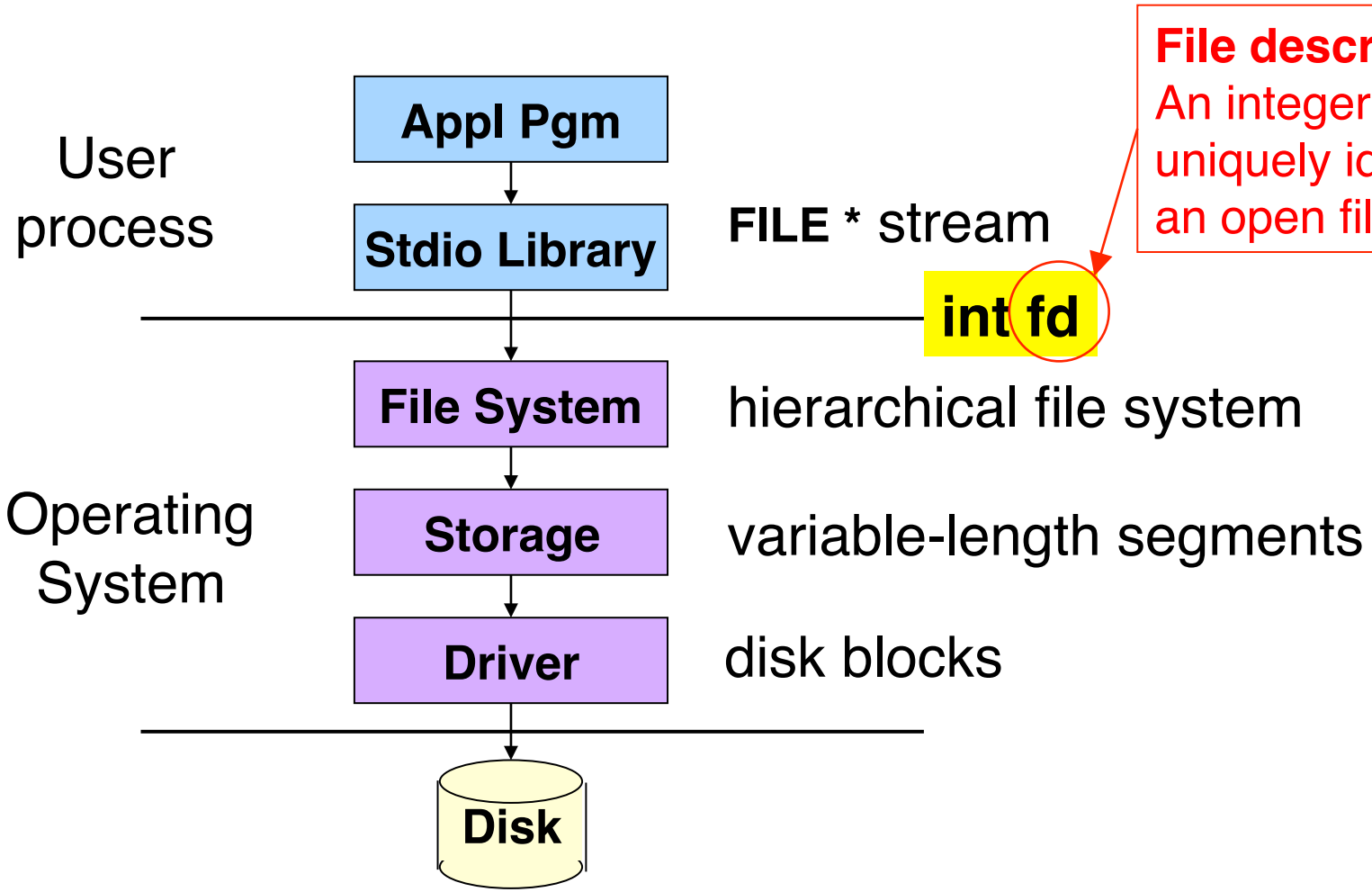
- `FILE *fopen("myfile.txt", "r")`
  - Open the named file and return a stream
  - Includes a mode, such as “r” for read or “w” for write
- Creates a FILE data structure for the file
  - Mode, status, buffer, …
  - Assigns fields and returns a pointer
- Opens or creates the file, based on the mode
  - Write (‘w’): create file with default permissions
  - Read (‘r’): open the file as read-only
  - Append (‘a’): open or create file, and seek to the end

Example: Formatted I/O

- `int fprintf(fp1, "Number: %d\n", i)`
  - Convert and write output to stream in specified format
- `int fscanf(fp1, "FooBar: %d", &i)`
  - Read from stream in format and assign converted values

- Specialized versions
  - `printf(...)` is just `fprintf(stdout, ...)`
  - `scanf(...)` is just `fscanf(stdin, ...)`

Layers of Abstraction



System-Level Functions for I/O

```
int creat(char *pathname, mode_t mode);
```

- Create a new file named **pathname**, and return a file descriptor

```
int open(char *pathname, int flags, mode_t mode);
```

- Open the file **pathname** and return a file descriptor

```
int close(int fd);
```

- Close **fd**

```
int read(int fd, void *buf, int count);
```

- Read up to **count** bytes from **fd** into the buffer at **buf**

```
int write(int fd, void *buf, int count);
```

- Writes up to **count** bytes into **fd** from the buffer at **buf**

```
int lseek(int fd, int offset, int whence);
```

- Assigns the file pointer of **fd** to a new value by applying an **offset**

Example: `open()`

- Converts a path name into a file descriptor
  - **`int open(const char *pathname, int flags, mode_t mode);`**
- Arguments
  - Pathname: name of the file
  - Flags: bit flags for **`O_RDONLY`**, **`O_WRONLY`**, **`O_RDWR`**
  - Mode: permissions to set if file must be created
- Returns
  - File descriptor (or a -1 if an error)
- Performs a variety of checks
  - E.g., whether the process is entitled to access the file
- Underlies **`fopen()`**

Example: `read()`

- Reads bytes from a file descriptor
  - `int read(int fd, void *buf, int count);`
- Arguments
  - File descriptor: integer descriptor returned by **`open()`**
  - Buffer: pointer to memory to store the bytes it reads
  - Count: maximum number of bytes to read
- Returns
  - Number of bytes read
    - Value of 0 if nothing more to read
    - Value of -1 if an error
- Performs a variety of checks
  - Whether file has been opened, whether reading is okay
- Underlies **`getchar()`** , **`fgets()`**, **`scanf()`** , etc.

Example: A Simple `getchar()`

```
int getchar(void) {
   char c;
   if (read(0, &c, 1) == 1)
      return c;
   else return EOF;
}
```

- Read one character from **`stdin`**
  - File descriptor `0` is **`stdin`**
  - **`&c`** points to the buffer
  - `1` is the number of bytes to read
- Read returns the number of bytes read
  - In this case, `1` byte means success

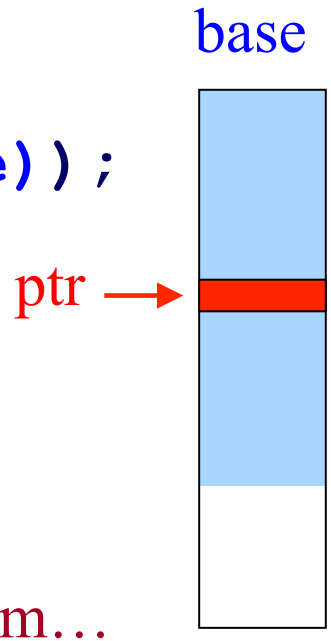
Making `getchar()` More Efficient

- Poor performance reading one byte at a time
  - Read system call is accessing the device (e.g., a disk)
  - Reading one byte from disk is very time consuming
  - Better to read and write in *larger chunks*
- Buffered I/O
  - Read a large chunk from disk into a buffer
    - Dole out bytes to the user process as needed
    - Discard buffer contents when the stream is closed
  - Similarly, for writing, write individual bytes to a buffer
    - And write to disk when full, or when stream is closed
    - Known as “flushing” the buffer

Better `getchar()` with Buffered I/O

```
int getchar(void) {
    static char base[1024];
    static char *ptr;
    static int cnt = 0;

    if (cnt--) return *ptr++;

    cnt = read(0, base, sizeof(base));
    if (cnt <= 0) return EOF;
    ptr = base;
    return getchar();
}
```

persistent variables



But, many functions may read (or write) the stream…

Details of FILE in stdio.h (K&R 8.5)

```
#define OPEN_MAX 20   /* max files open at once */

typedef struct _iobuf {
    int  cnt;      /* num chars left in buffer */
    char *ptr;     /* ptr to next char in buffer */
    char *base;    /* beginning of buffer */
    int  flag;     /* open mode flags, etc. */
    char fd;       /* file descriptor */
} FILE;
extern FILE _iob[OPEN_MAX];

#define stdin  (&_iob[0])
#define stdout (&_iob[1])
#define stderr (&_iob[2])
```

A Funny Thing About Buffered I/O

- The standard library also buffers **output**; example:

```
int main(void) {
   printf("Step 1\n");
   sleep(10);
   printf("Step 2\n");
   return 0;
}
```

- Run "a.out > out.txt &" and then "tail -f out.txt"
  - To run `a.out` in the background, outputting to `out.txt`
  - And then to see the contents on `out.txt`
- Neither line appears till ten seconds have elapsed
  - Because the output is being buffered
  - Add `fflush(stdout)` to flush the output buffer
  - `fclose()` also flushes the buffer before closing

Summary on I/O Management

- System-level I/O functions provide abstractions
  - Stream as a source or destination of data
  - Functions for manipulating streams
- Standard I/O library builds on these abstractions
  - Calls system-level functions for low-level I/O
  - Adds buffering
- Powerful examples of abstraction
  - Applications interact with streams at a high level
  - Standard I/O library interact with streams at lower level
  - Only the OS deals with the device-specific details

Process Management

Goals of this Lecture: Processes

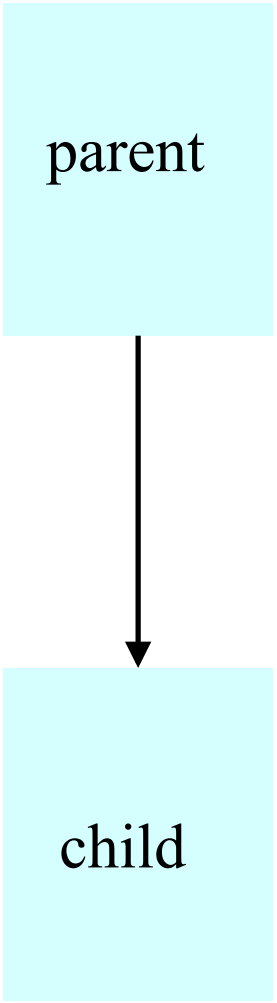
- Help you learn about:
  - Creating new processes
  - Programmatically redirecting stdin, stdout, and stderr
  - (Appendix) communication between processes via pipes
- Why?
  - Creating new processes and programmatic redirection are fundamental tasks of a Unix **shell** (Assignment 7)

Why Create a New Process?

- Run a new program
  - E.g., shell executing a program entered at command line
  - Or, even running an entire pipeline of commands
  - Such as "`wc -l * | sort | uniq -c | sort -nr`"
- Run a new thread of control for the same program
  - E.g., a Web server handling a new Web request
  - While continuing to allow more requests to arrive
  - Essentially time sharing the computer
- Underlying mechanism
  - A process executes **`fork()`** to create a child process
  - (Optionally) child does **`exec()`** of a new program

Creating a New Process

- Cloning an existing process
  - Parent process creates a new child process
  - The two processes then run concurrently
- Child process inherits state from parent
  - Identical (but separate) copy of virtual address space
  - Copy of the parent's open file descriptors
  - Parent and child share access to open files
- Child then runs independently
  - Executing independently, including invoking a new program
  - Reading and writing its own address space



Fork System-Level Function

- `fork()` is called once
  - But returns twice, once in each process
- Telling which process is which
  - Parent: `fork()` returns the child's process ID
  - Child: `fork()` returns 0

```
pid = fork();
if (pid != 0) {
    /* in parent */
   …
} else {
    /* in child */
    …
}
```

Fork and Process State

- Inherited
  - User and group IDs
  - Signal handling settings
  - Stdio
  - File pointers
  - Root directory
  - File mode creation mask
  - Resource limits
  - Controlling terminal
  - All machine register states
  - Control register(s)
  - …
- Separate in child
  - Process ID
  - Address space (memory)
  - File descriptors
  - Parent process ID
  - Pending signals
  - Time signal reset times
  - …

Example: What Output?

```
int main(void)
{
    pid_t pid;
    int x = 1;

    pid = fork();
    if (pid != 0) {
        printf("parent: x = %d\n", --x);
        exit(0);
    } else {
        printf("child: x = %d\n", ++x);
        exit(0);
    }
}
```

Executing a New Program

- `fork()` copies the state of the parent process
  - Child continues running the parent program
  - … with a copy of the process memory and registers
- Need a way to invoke a new program
  - In the context of the newly-created child process
- Example

NULL-terminated array
Contains command-line arguments
(to become "argv[]" of ls)

program

```
execvp("ls", argv);
fprintf(stderr, "exec failed\n");
exit(EXIT_FAILURE);
```

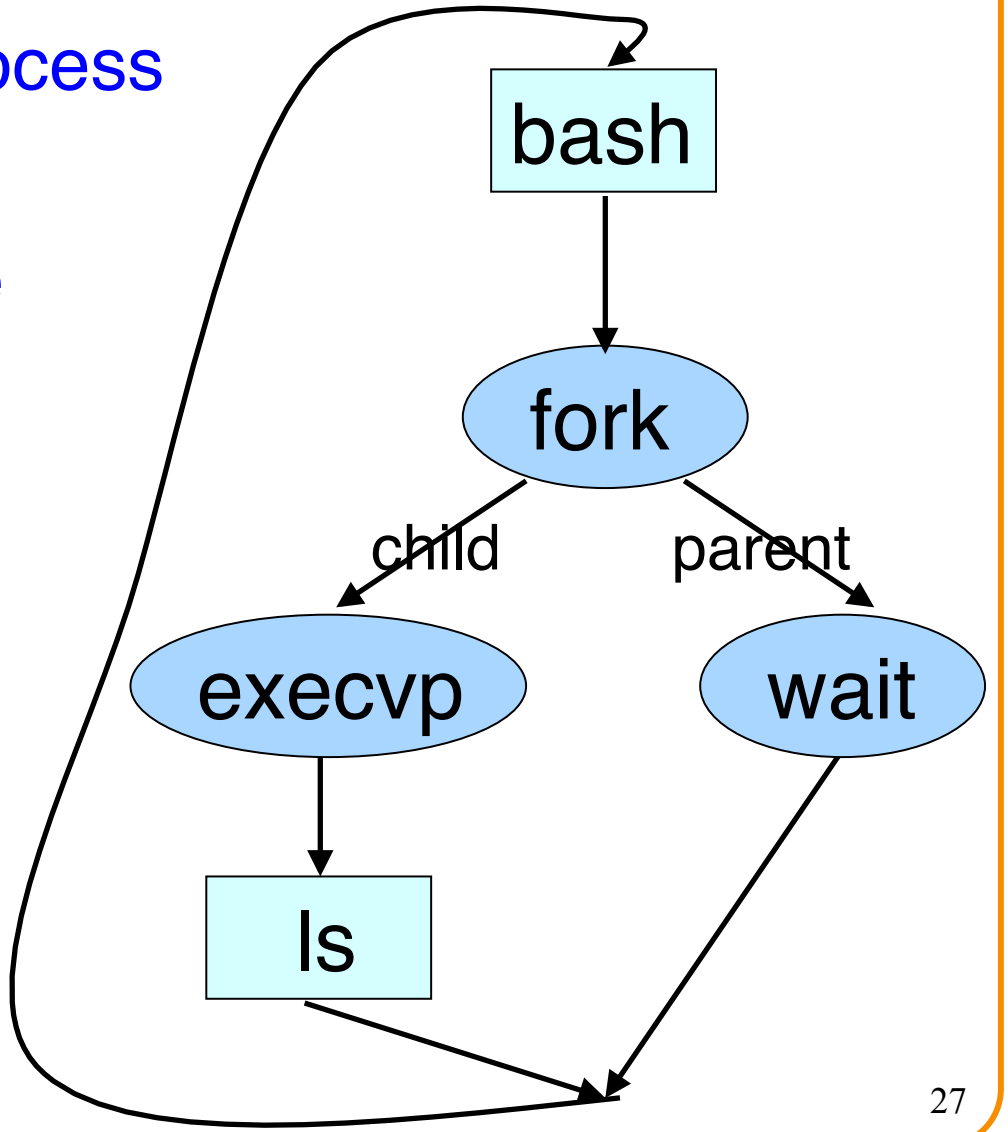
Waiting for the Child to Finish

- Parent should wait for children to finish
  - Example: a shell waiting for operations to complete
- Waiting for a child to terminate: **`wait()`**
  - Blocks until some child terminates
  - Returns the process ID of the child process
  - Or returns -1 if no children exist (i.e., already exited)
- Waiting for specific child to terminate: **`waitpid()`**
  - Blocks till a child with particular process ID terminates

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

Example: A Simple Shell

- Shell is the parent process
  - E.g., bash
- Parses command line
  - E.g., "ls –l"
- Invokes child process
  - **`fork()`**, **`execvp()`**
- Waits for child
  - **`wait()`**



Simple Shell Code

```
Parse command line
Assign values to somepgm, someargv
pid = fork();
if (pid == 0) {
    /* in child */
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
Repeat the previous
```

Simple Shell Trace (1)

Parent Process

```
Parse command line
Assign values to somepgm, someargv
pid = fork();
if (pid == 0) {
   /* in child */
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
Repeat the previous
```

Parent reads and parses command line
Parent assigns values to **`somepgm`** and **`someargv`**

Simple Shell Trace (2)

Parent Process

```
Parse command line
Assign values to somepgm, someargv
pid = fork();
if (pid == 0) {
   /* in child */
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
Repeat the previous
```

executing concurrently

Child Process

```
Parse command line
Assign values to somefile, someargv
pid = fork();
if (pid == 0) {
   /* in child */
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
Repeat the previous
```

`fork()` creates child process

Which process gets the CPU first? Let's assume the parent…

Simple Shell Trace (3)

Parent Process

child's pid

Child Process

```
Parse command line
Assign values to somepgm, someargv
pid = fork();
if (pid == 0) {
   /* in child */
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
Repeat the previous
```

executing concurrently

```
Parse command line
Assign values to somefile, someargv
pid = fork();
if (pid == 0) {
   /* in child */
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
Repeat the previous
```

In parent, pid != 0; parent waits; OS gives CPU to child

Simple Shell Trace (4)

Parent Process

```
Parse command line
Assign values to somepgm, someargv
pid = fork();
if (pid == 0) {
   /* in child */
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
Repeat the previous
```

0

executing concurrently

Child Process

```
Parse command line
Assign values to somefile, someargv
pid = fork();
if (pid == 0) {
   /* in child */
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
Repeat the previous
```

In child, pid == 0; child calls **`execvp()`**

Simple Shell Trace (5)

Parent Process

```
Parse command line
Assign values to somepgm, someargv
pid = fork();
if (pid == 0) {
   /* in child */
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
Repeat the previous
```

executing concurrently

Child Process

```
somepgm
```

In child, somepgm overwrites shell program;
`main()` is called with someargv as argv parameter

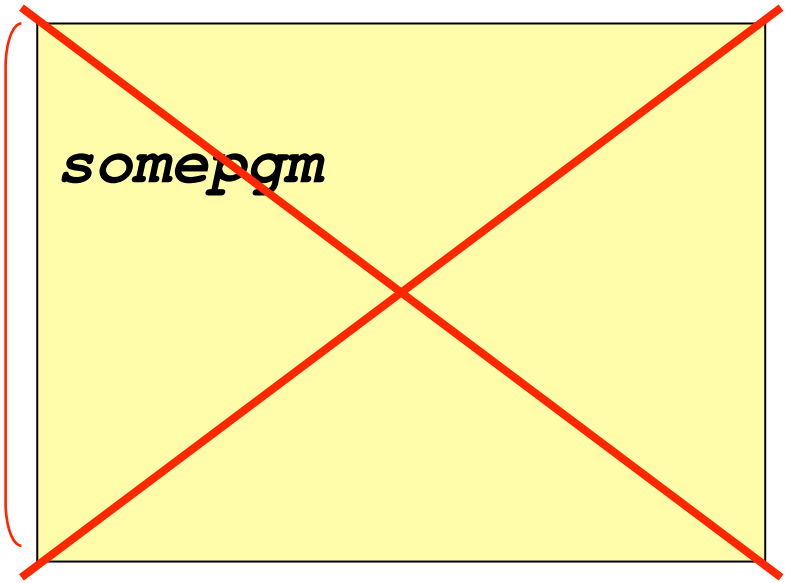
Simple Shell Trace (6)

Parent Process

```
Parse command line
Assign values to somepgm, someargv
pid = fork();
if (pid == 0) {
   /* in child */
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
Repeat the previous
```

executing concurrently

Child Process



Somepgm executes in child, and eventually exits

Simple Shell Trace (7)

Parent Process

```
Parse command line
Assign values to somepgm, someargv
pid = fork();
if (pid == 0) {
   /* in child */
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
Repeat the previous
```

Parent returns from **`wait()`** and proceeds

Combined Fork/Exec/Wait

- Common combination of operations
  - **`fork()`** to create a new child process
  - **`exec()`** to invoke new program in child process
  - **`wait()`** in the parent process for the child to complete
- Single call that combines all three
  - **`int system(const char *cmd);`**
- Example

```
int main(void) {
   system("echo Hello world");
   return 0;
}
```

Fork and Virtual Memory

- Incidentally…
- Question:
  - `fork()` duplicates an entire process (text, bss, data, rodata, stack, heap sections)
  - Isn't that *very* inefficient???!!!
- Answer:
  - Using virtual memory, not really!
  - Upon `fork()`, OS creates virtual pages for child process
  - Each child virtual page points to real page (in memory or on disk) of parent
  - OS duplicates real pages incrementally, and only if/when "write" occurs

Redirection

- Unix allows redirection of stdin, stdout, or stderr
- How?
  - Use **`open()`**, **`creat()`**, and **`close()`** system calls
    - Described in **I/O Management** lecture
  - Use **`dup()`** system call…

```
int dup(int oldfd);
```

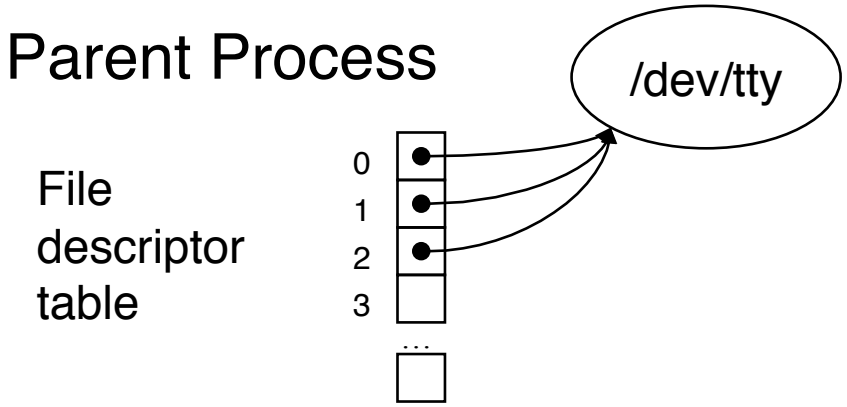
- Create a copy of the file descriptor oldfd. After a successful return from dup() or dup2(), the old and new file descriptors may be used interchangeably. They refer to the same open file description and thus share file offset and file status flags. Uses the lowest-numbered unused descriptor for the new descriptor. Return the new descriptor, or -1 if an error occurred.

Redirection Example

How does shell implement “somepgm > somefile”?

```
pid = fork();
if (pid == 0) {
    /* in child */
    fd = creat("somefile", 0640);
    close(1);
    dup(fd);
    close(fd);
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

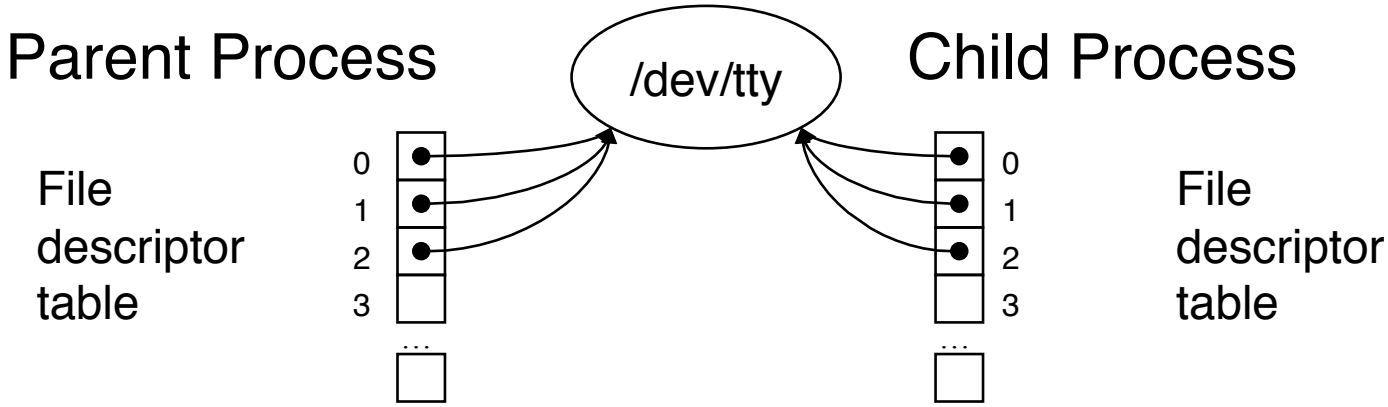
Redirection Example Trace (1)




```
pid = fork();
if (pid == 0) {
   /* in child */
   fd = creat("somefile", 0640);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

Parent has file descriptor table; first three point to "terminal"

Redirection Example Trace (2)

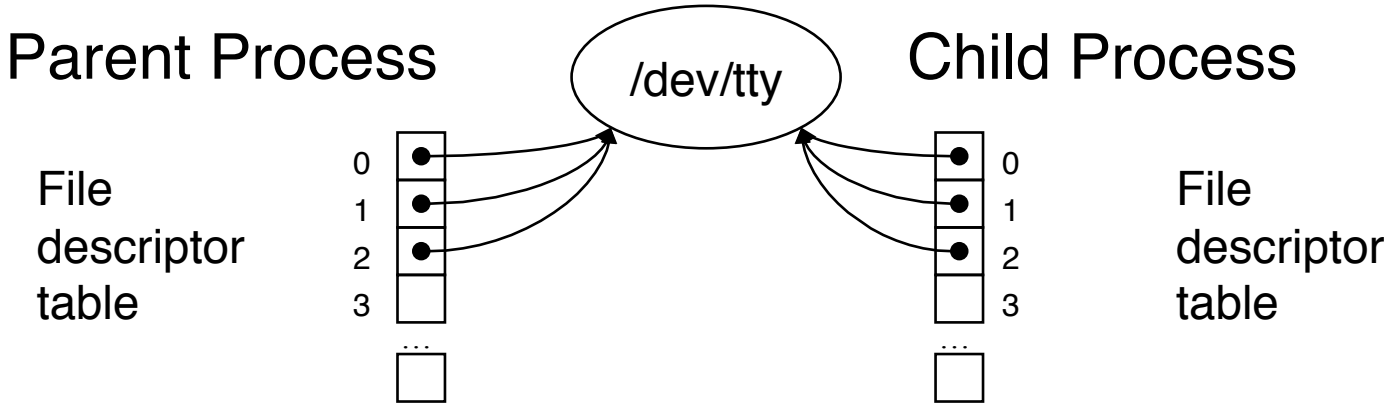



```
pid = fork();
if (pid == 0) {
   /* in child */
   fd = creat("somefile", 0640);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

```
pid = fork();
if (pid == 0) {
   /* in child */
   fd = creat("somefile", 0640);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

Parent forks child; child has identical file descriptor table

Redirection Example Trace (3)

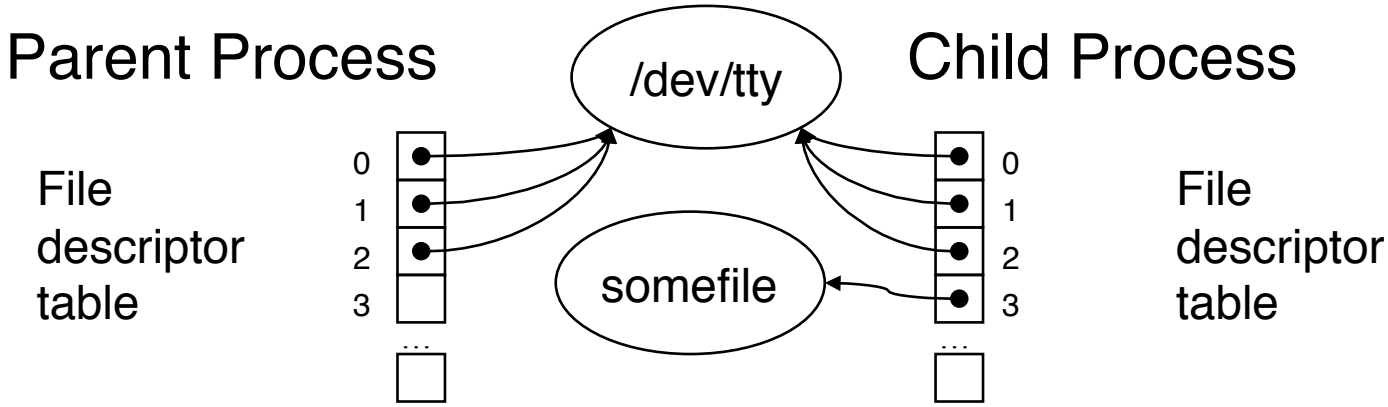



```
pid = fork();
if (pid == 0) {
   /* in child */
   fd = creat("somefile", 0640);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

```
pid = fork();
if (pid == 0) {
   /* in child */
   fd = creat("somefile", 0640);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

Let's say parent gets CPU first; parent waits

Redirection Example Trace (4)



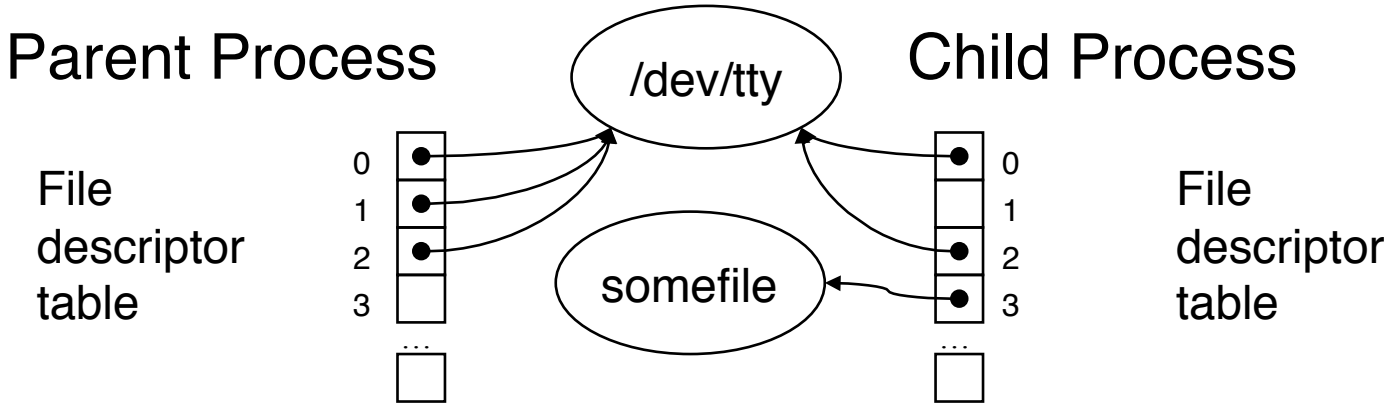

```
pid = fork();
if (pid == 0) {
   /* in child */
   fd = creat("somefile", 0640);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

3

```
pid = fork();
if (pid == 0) {
   /* in child */
   fd = creat("somefile", 0640);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

Child gets CPU; child creates somefile

Redirection Example Trace (5)



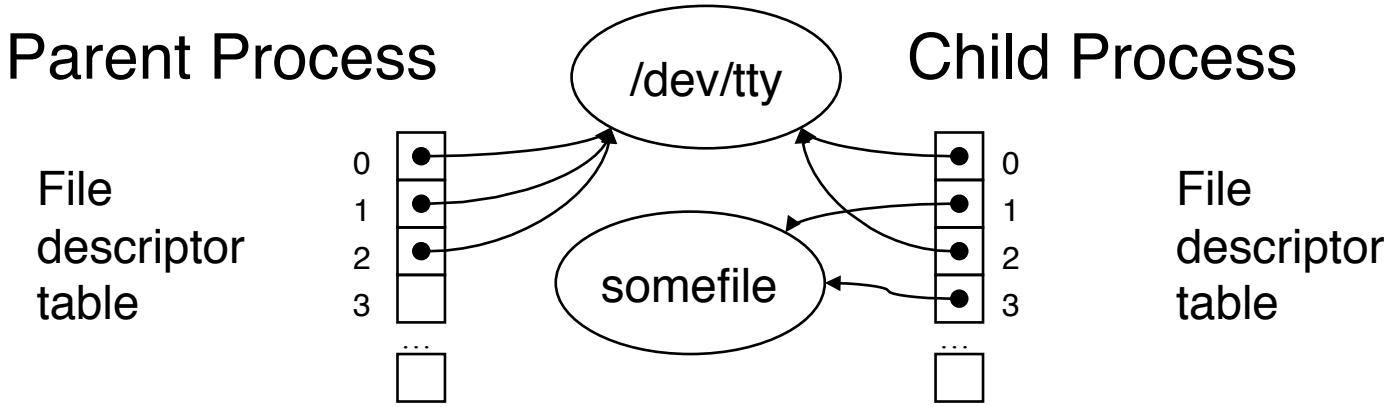

```
pid = fork();
if (pid == 0) {
   /* in child */
   fd = creat("somefile", 0640);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

3

```
pid = fork();
if (pid == 0) {
   /* in child */
   fd = creat("somefile", 0640);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

Child closes file descriptor 1 (stdout)

Redirection Example Trace (6)



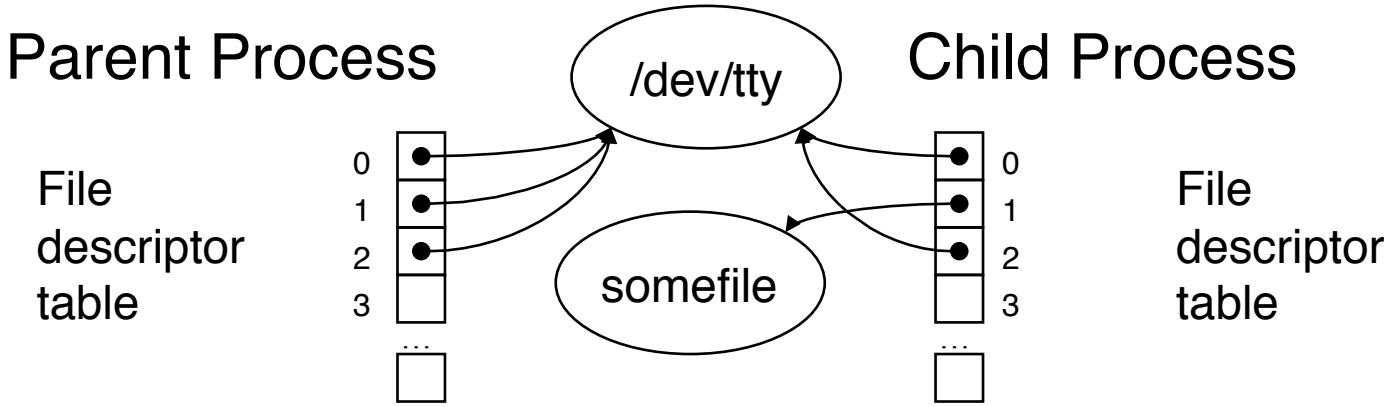

```
pid = fork();
if (pid == 0) {
   /* in child */
   fd = creat("somefile", 0640);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

3

```
pid = fork();
if (pid == 0) {
   /* in child */
   fd = creat("somefile", 0640);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

Child duplicates file descriptor 3 into first unused spot

Redirection Example Trace (7)



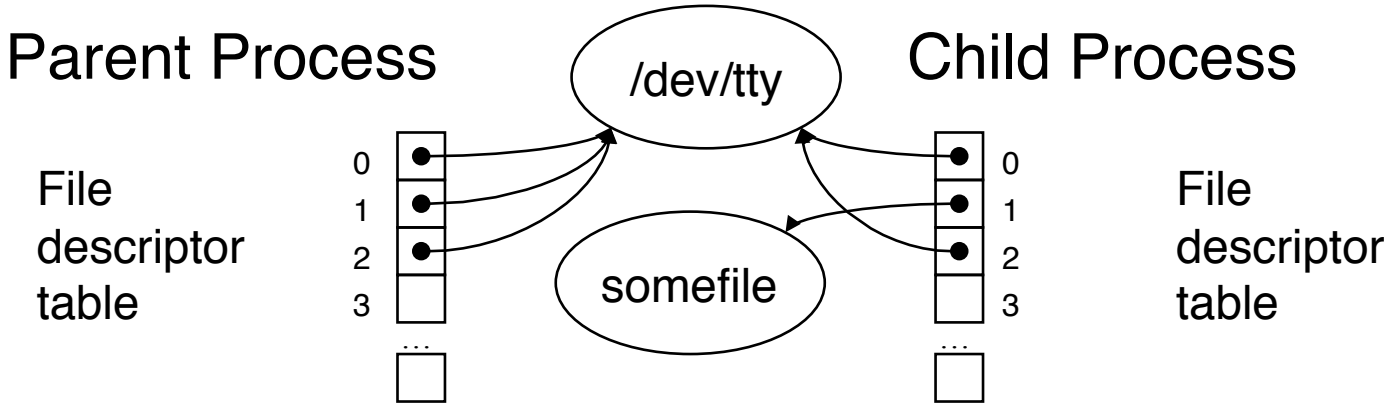

```
pid = fork();
if (pid == 0) {
   /* in child */
   fd = creat("somefile", 0640);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

3

```
pid = fork();
if (pid == 0) {
   /* in child */
   fd = creat("somefile", 0640);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

Child closes file descriptor 3

Redirection Example Trace (8)



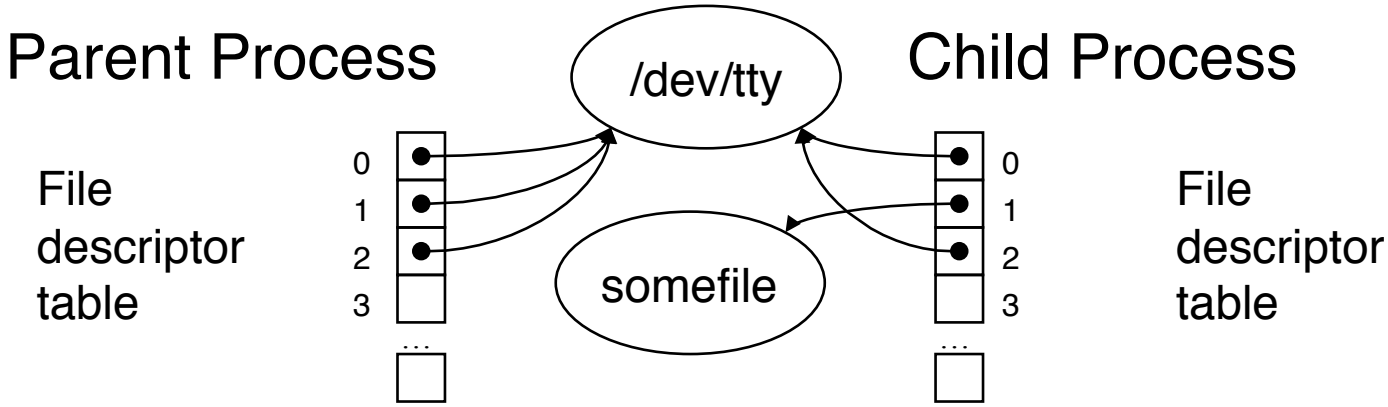

```
pid = fork();
if (pid == 0) {
   /* in child */
   fd = creat("somefile", 0640);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

3

```
pid = fork();
if (pid == 0) {
   /* in child */
   fd = creat("somefile", 0640);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

Child calls execvp()

Redirection Example Trace (9)

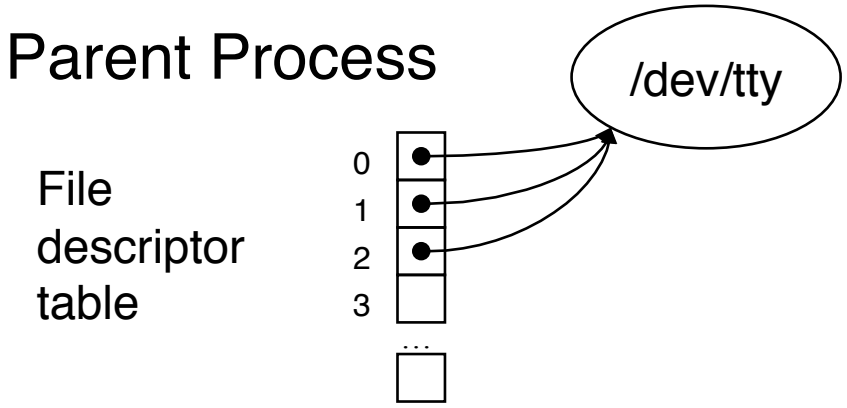



```
pid = fork();
if (pid == 0) {
   /* in child */
   fd = creat("somefile", 0640);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepfm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

somepgm

Somepgm executes with stdout redirected to somefile

Redirection Example Trace (10)




```
pid = fork();
if (pid == 0) {
   /* in child */
   fd = creat("somefile", 0640);
   close(1);
   dup(fd);
   close(fd);
   execvp(somefile, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

Somepgm exits; parent returns from **`wait()`** and proceeds

The Beginnings of a Unix Shell

- A shell is mostly a big loop
  - Parse command line from stdin
  - Expand wildcards (‘*’)
  - Interpret redirections (‘<‘, and ‘>’)
  - **`fork()`**, **`dup()`**, **`exec()`**, and **`wait()`**, as necessary
- Start from the code in earlier slides
  - And edit till it becomes a Unix shell
  - This is the heart of the last programming assignment

Summary

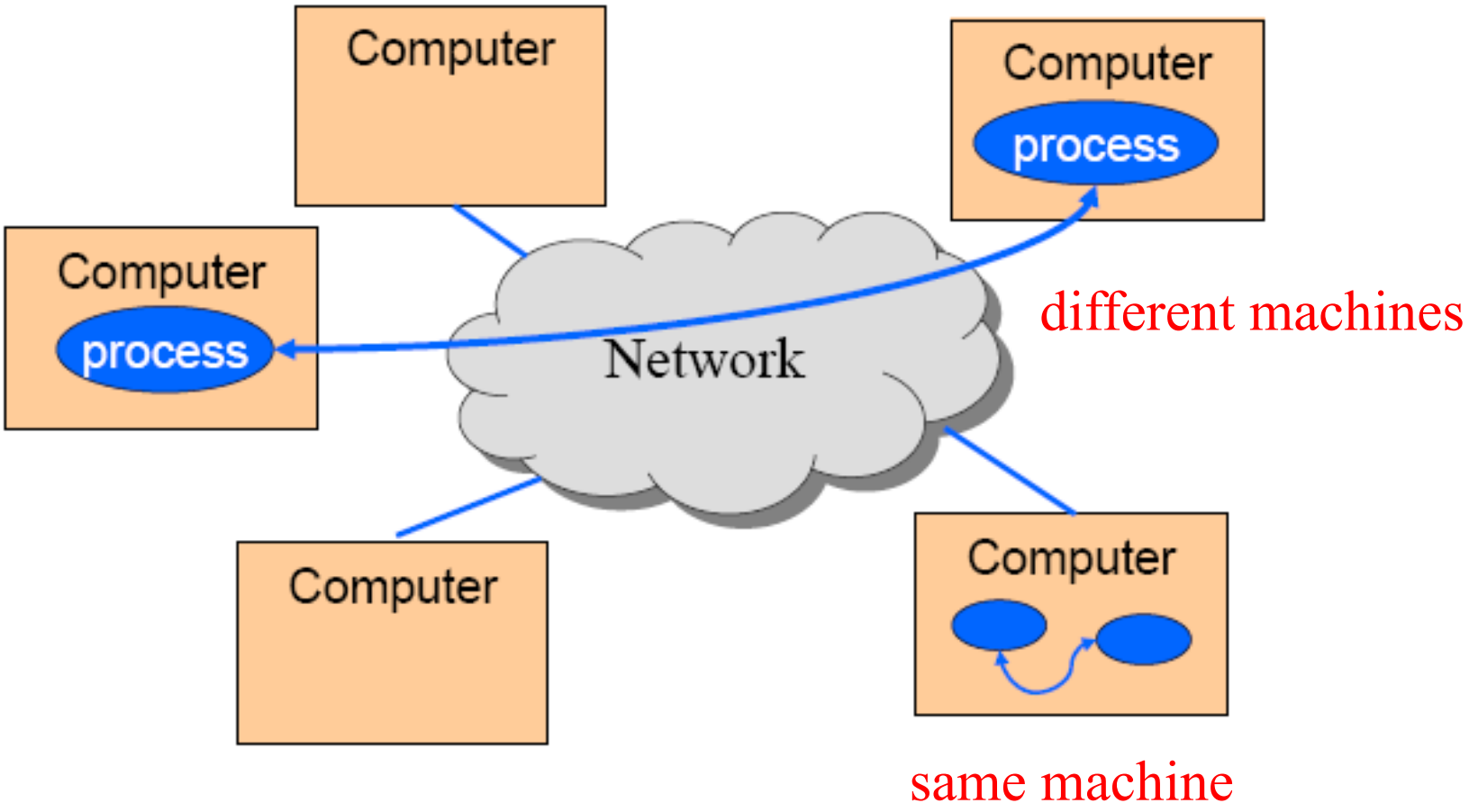
- System-level functions for creating processes
  - **`fork()`**: process creates a new child process
  - **`wait()`**: parent waits for child process to complete
  - **`exec()`**: child starts running a new program
  - **`system()`**: combines fork, wait, and exec all in one
- System-level functions for redirection
  - **`open()`** / **`creat()`**: to open a file descriptor
  - **`close()`**: to close a file descriptor
  - **`dup()`**: to duplicate a file descriptor

Appendix

Inter-Process Communication (IPC)

IPC

- Mechanism by which two processes exchange information and coordinate activities



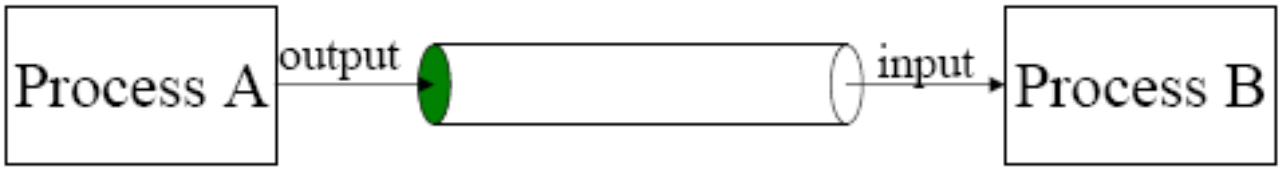
IPC Mechanisms

- Pipes
  - Processes on the same machine
  - Allows parent process to communicate with child process
  - Allows two "sibling" processes to communicate
  - Used mostly for a pipeline of filters
- Sockets (see COS 461)
  - Processes on any machines
  - Processes created independently
  - Used for client/server communication (e.g., Web)

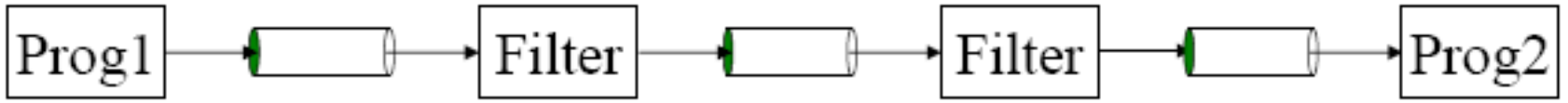
Both provide abstraction of an "ordered stream of bytes"

Pipes

- Provides an interprocess communication channel



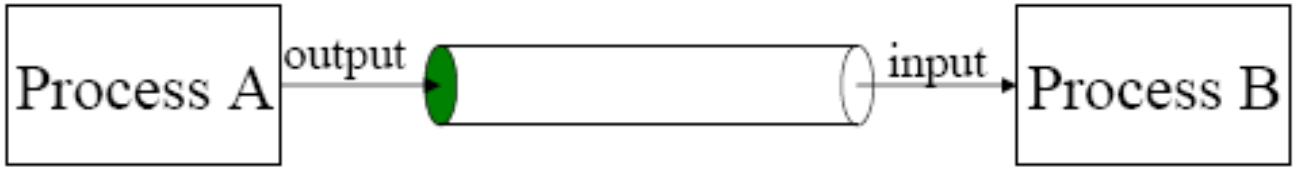
- A filter is a process that reads from `stdin` and writes to `stdout`



Example Use of Pipes

- Compute a histogram of content types in my e-mail
  - Many e-mail messages, consisting of many lines
  - Lines like “Content-Type: image/jpeg” indicate the type
- Pipeline of Unix commands
  - Identifying content type: **`grep -i Content-Type *`**
  - Extracting just the type: **`cut -d" " -f2`**
  - Sorting the list of types: **`sort`**
  - Counting the unique types: **`uniq -c`**
  - Sorting the counts: **`sort –nr`**
- Simply running this at the shell prompt:
  - **`grep -i Content-Type * | cut -d" " -f2 | sort | uniq -c | sort –nr`**

Creating a Pipe



- Pipe is a communication channel abstraction
  - Process A can write to one end using "write" system call
  - Process B can read from the other end using "read" system call
- System call

```
int pipe( int fd[2] );
return 0 upon success –1 upon failure
fd[0] is open for reading
fd[1] is open for writing
```

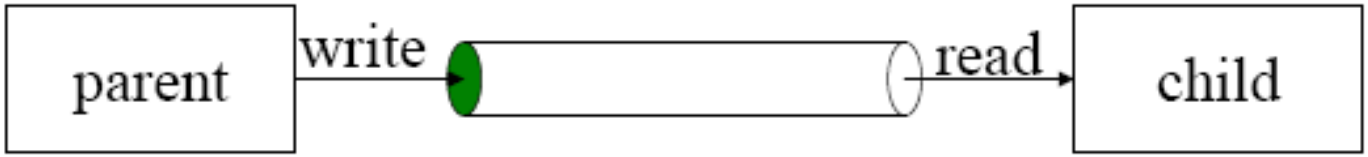
- Two coordinated processes created by `fork` can pass data to each other using a pipe.

Pipe Example

```
int pid, p[2];
...
if (pipe(p) == -1)
    exit(1);
pid = fork();
if (pid == 0) {
    close(p[1]);
    ... read using p[0] as fd until EOF ...
}
else {
    close(p[0]);
    ... write using p[1] as fd ...
    close(p[1]); /* sends EOF to reader */
    wait(&status);
}
```

child

parent

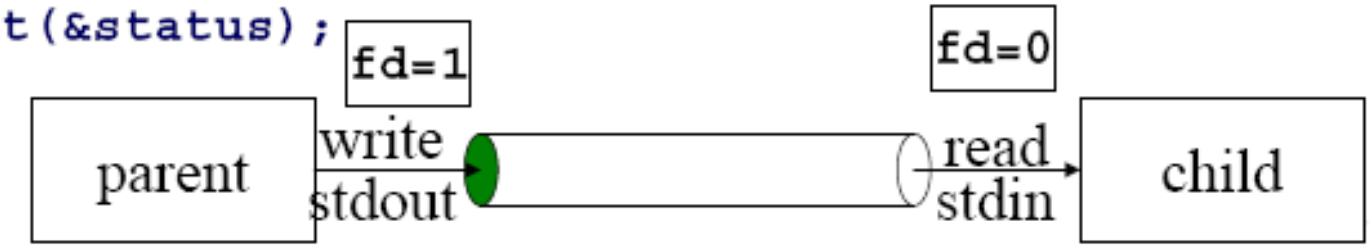


Pipes and Stdio

```
int pid, p[2];
if (pipe(p) == -1)
   exit(1);
pid = fork();
if (pid == 0) {
   close(p[1]);
   dup2(p[0],0);
   close(p[0]);
   ... read from stdin ...
}
else {
   close(p[0]);
   dup2(p[1],1);
   close(p[1]);
   ... write to stdout ...
   wait(&status);
}
```

child makes stdin (0)
the read side of the pipe

parent makes stdout (1)
the write side of the pipe

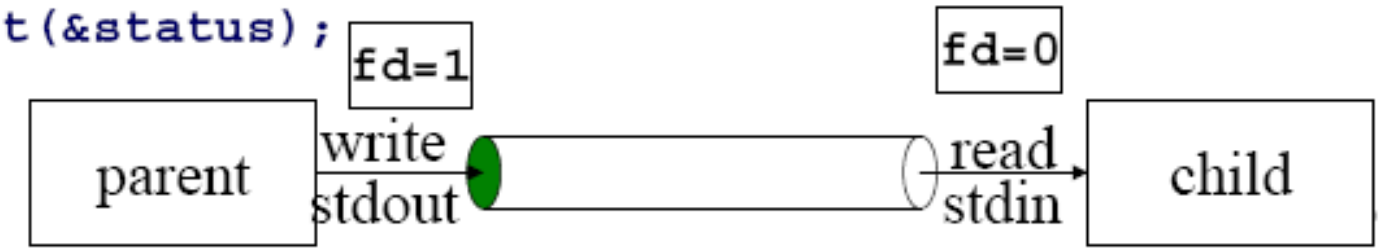


Pipes and Exec

```
int pid, p[2];
if (pipe(p) == -1)
    exit(1);
pid = fork();
if (pid == 0) {
    close(p[1]);
    dup2(p[0],0);
    close(p[0]);
    execl(...);
}
else {
    close(p[0]);
    dup2(p[1],1);
    close(p[1]);
    ... write to stdout ...
    wait(&status);
}
```

child process

invokes a new program



Signals

Jennifer Rexford

Goals of this Lecture

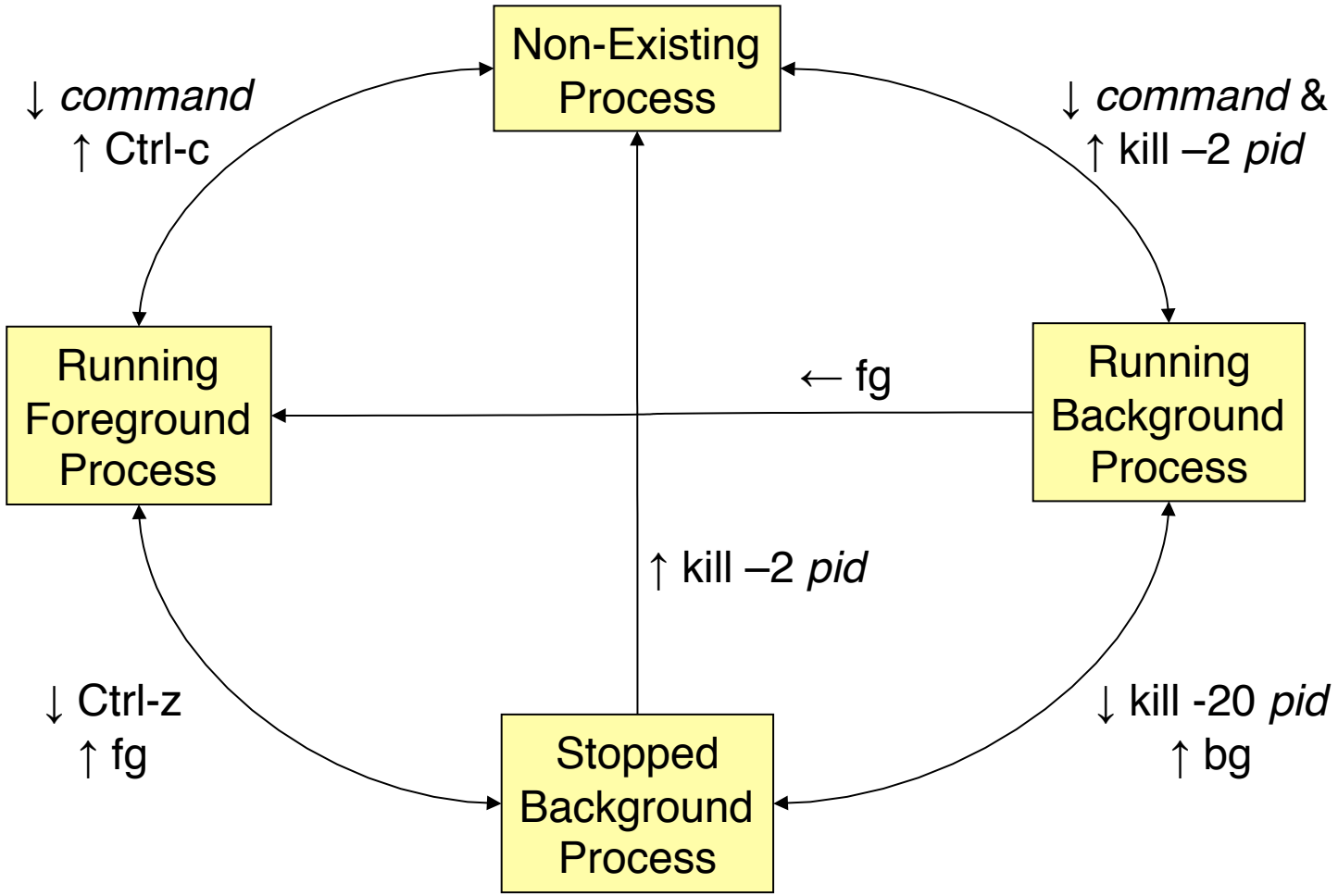
- Help you learn about:
  - Sending signals
  - Handling signals

- … and thereby …
  - How the OS exposes the occurrence of some exceptions to application processes
  - How application processes can control their behavior in response to those exceptions

Outline

1. **Unix Process Control**
2. Signals
3. Sending Signals
4. Handling Signals
5. Race Conditions and Critical Sections
6. Blocking Signals
7. Alarms
8. (If time) Interval Timers
9. Conclusion

Unix Process Control



Unix Process Control

[Demo of Unix process control using infloop.c]

Process Control Implementation

Exactly what happens when you:

- Type Ctrl-c?
  - Keystroke generates **interrupt**
  - OS handles interrupt
  - OS sends a 2/SIGINT **signal**
- Type Ctrl-z?
  - Keystroke generates **interrupt**
  - OS handles interrupt
  - OS sends a 20/SIGTSTP **signal**

Recall "Exceptions and Processes" lecture

Process Control Implementation (cont.)

Exactly what happens when you:

- Issue a "**`kill -sig pid`**" command?
  - **`kill`** command executes **trap**
  - OS handles trap
  - OS sends a *sig* **signal** to the process whose id is *pid*
- Issue a "**`fg`**" or "**`bg`**" command?
  - **`fg`** or **`bg`** command executes **trap**
  - OS handles trap
  - OS sends a 18/SIGCONT **signal** (and does some other things too!)

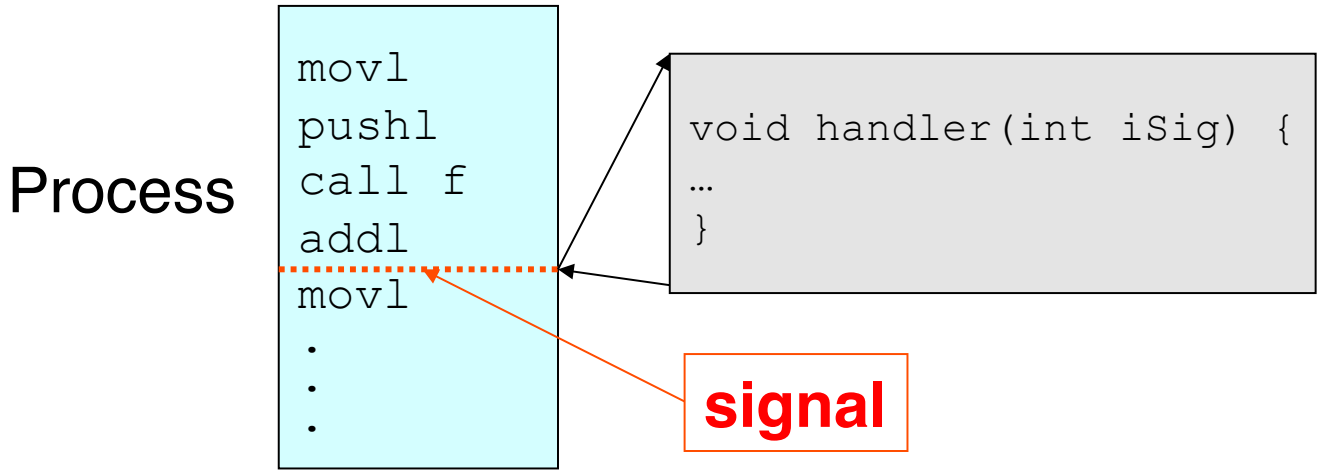
Recall "Exceptions and Processes" lecture

Outline

1. Unix Process Control
2. **Signals**
3. Sending Signals
4. Handling Signals
5. Race Conditions and Critical Sections
6. Blocking Signals
7. Alarms
8. (If time) Interval Timers
9. Conclusion

Signal: Notification of an Event

- Exception occurs (interrupt, trap, fault, or abort)
  - Context switches to OS
- OS sends signal to application process
  - Sets a bit in a vector indicating that a signal of type X occurred
- Process regains CPU and default action for signal executes
  - Can install a **signal handler** to change action
- (Optionally) Application process resumes where it left off



Examples of Signals

User types Ctrl-c

- Interrupt occurs
- Context switches to OS
- OS sends 2/SIGINT signal to application process
- Default action for 2/SIGINT signal is “terminate”

Process makes illegal memory reference

- Fault occurs
- Context switches to OS
- OS sends 11/SIGSEGV signal to application process
- Default action for 11/SIGSEGV signal is “terminate”

Outline

1. Unix Process Control
2. Signals
3. **Sending Signals**
4. Handling Signals
5. Race Conditions and Critical Sections
6. Blocking Signals
7. Alarms
8. (If time) Interval Timers
9. Conclusion

Sending Signals via Keystrokes

Three signals can be sent from keyboard:

- **Ctrl-c** → 2/SIGINT signal
  - Default action is "terminate"
- **Ctrl-z** → 20/SIGTSTP signal
  - Default action is "stop until next 18/SIGCONT"
- **Ctrl-** → 3/SIGQUIT signal
  - Default action is "terminate"

Sending Signals via Commands

kill Command

`kill -`*`signal pid`*

- Send a signal of type ***`signal`*** to the process with id ***`pid`***
- No signal type name or number specified => sends 15/SIGTERM signal
- Default action for 15/SIGTERM is “terminate”
- Editorial: Better command name would be **`sendsig`**

Examples

```
kill -2 1234
kill -SIGINT 1234
```

- Same as pressing Ctrl-c if process 1234 is running in foreground

Sending Signals via Function Calls

`raise()`

```
int raise(int iSig);
```

- Commands OS to send a signal of type `iSig` to current process
- Returns 0 to indicate success, non-0 to indicate failure

Example

```
int iRet = raise(SIGINT); /* Process commits suicide. */
assert(iRet != 0);        /* Shouldn't get here. */
```

Sending Signals via Function Calls

`kill()`

```
int kill(pid_t iPid, int iSig);
```

- Sends a `iSig` signal to the process whose id is `iPid`
- Equivalent to `raise(iSig)` when `iPid` is the id of current process
- Editorial: Better function name would be `sendsig()`

Example

```
pid_t iPid = getpid();          /* Process gets its id.*/
int iRet = kill(iPid, SIGINT); /* Process sends itself a
assert(iRet != 0);                SIGINT signal (commits
                                  suicide) */
```

Outline

1. Unix Process Control
2. Signals
3. Sending Signals
4. **Handling Signals**
5. Race Conditions and Critical Sections
6. Blocking Signals
7. Alarms
8. (If time) Interval Timers
9. Conclusion

Handling Signals

Each signal type has a default action

- For most signal types, default action is “terminate”

A program can **install** a **signal handler** to change action of (almost) any signal type

Uncatchable Signals

Special cases: A program *cannot* install a signal handler for signals of type:

- 9/SIGKILL
  - Default action is “terminate”
- 19/SIGSTOP
  - Default action is “stop until next 18/SIGCONT”

Installing a Signal Handler

`signal()`

```
sighandler_t signal(int iSig,
                    sighandler_t pfHandler);
```

- Installs function **`pfHandler`** as the handler for signals of type **`iSig`**
- **`pfHandler`** is a function pointer:

```
typedef void (*sighandler_t)(int);
```

- Returns the old handler on success, **`SIG_ERR`** on error
- After call, **`(*pfHandler)`** is invoked whenever process receives a signal of type **`iSig`**

Installing a Handler Example 1

Program testsignal.c:

```
#define _GNU_SOURCE /* Use modern handling style */
#include <stdio.h>
#include <assert.h>
#include <signal.h>

static void myHandler(int iSig) {
   printf("In myHandler with argument %d\n", iSig);
}
…
```

Installing a Handler Example 1 (cont.)

Program testsignal.c (cont.):

```
…
int main(void) {
    void (*pfRet)(int);
    pfRet = signal(SIGINT, myHandler);
    assert(pfRet != SIG_ERR);

    printf("Entering an infinite loop\n");
    for (;;)
        ;
    return 0;
}
```

Installing a Handler Example 1 (cont.)

[Demo of testsignal.c]

Installing a Handler Example 2

Program testsignalall.c:

```
#define _GNU_SOURCE
#include <stdio.h>
#include <assert.h>
#include <signal.h>

static void myHandler(int iSig) {
   printf("In myHandler with argument %d\n", iSig);
}
…
```

Installing a Handler Example 2 (cont.)

Program testsignalall.c (cont.):

```
…
int main(void) {
   void (*pfRet)(int);
   pfRet = signal(SIGHUP,  myHandler);   /* 1 */
   pfRet = signal(SIGINT,  myHandler);   /* 2 */
   pfRet = signal(SIGQUIT, myHandler);   /* 3 */
   pfRet = signal(SIGILL,  myHandler);   /* 4 */
   pfRet = signal(SIGTRAP, myHandler);   /* 5 */
   pfRet = signal(SIGABRT, myHandler);   /* 6 */
   pfRet = signal(SIGBUS,  myHandler);   /* 7 */
   pfRet = signal(SIGFPE,  myHandler);   /* 8 */
   pfRet = signal(SIGKILL, myHandler);   /* 9 */
   …
```

This call fails

Installing a Handler Example 2 (cont.)

Program testsignalall.c (cont.):

```
    …
    /* Etc., for every signal. */

    printf("Entering an infinite loop\n");
    for (;;)
        ;
    return 0;
}
```

Installing a Handler Example 2 (cont.)

[Demo of testsignalall.c]

Installing a Handler Example 3

Program generates lots of temporary data

- Stores the data in a temporary file
- Must delete the file before exiting

```
…
int main(void) {
    FILE *psFile;
    psFile = fopen("temp.txt", "w");
    …
    fclose(psFile);
    remove("temp.txt");
    return 0;
}
```

Example 3 Problem

What if user types Ctrl-c?

- OS sends a 2/SIGINT signal to the process
- Default action for 2/SIGINT is “terminate”

Problem: The temporary file is not deleted

- Process terminates before `remove("temp.txt")` is executed

Challenge: Ctrl-c could happen at any time

- Which line of code will be interrupted???

Solution: Install a signal handler

- Define a “clean up” function to delete the file
- Install the function as a signal handler for 2/SIGINT

Example 3 Solution

```
…
static FILE *psFile; /* Must be global. */
static void cleanup(int iSig) {
   fclose(psFile);
   remove("temp.txt");
   exit(0);
}
int main(void) {
   void (*pfRet)(int);
   psFile = fopen("temp.txt", "w");
   pfRet = signal(SIGINT, cleanup);
   …
   cleanup(0);  /* or raise(SIGINT); */
   return 0;  /* Never get here. */
}
```

SIG_IGN

Predefined value: **SIG_IGN**

Can use as argument to **`signal()`** to **ignore** signals

```
int main(void) {
    void (*pfRet)(int);
    pfRet = signal(SIGINT, SIG_IGN);
    assert(pfRet != SIG_ERR);
    …
}
```

Subsequently, process will ignore 2/SIGINT signals

SIG_DFL

Predefined value: **SIG_DFL**

Can use as argument to **`signal()`** to **restore default action**

```
int main(void) {
   void (*pfRet)(int);
   …
   pfRet = signal(SIGINT, somehandler);
   assert(pfRet != SIG_ERR);
   …
   pfRet = signal(SIGINT, SIG_DFL);
   assert(pfRet != SIG_ERR);

   …
}
```

Subsequently, process will handle 2/SIGINT signals using default action for 2/SIGINT signals (“terminate”)

Outline

1. Unix Process Control
2. Signals
3. Sending Signals
4. Handling Signals
5. **Race Conditions and Critical Sections**
6. Blocking Signals
7. Alarms
8. (If time) Interval Timers
9. Conclusion

Race Conditions and Critical Sections

Race Condition

A flaw in a program whereby the correctness of the program is critically dependent on the sequence or timing of events beyond the program's control

Critical Section

A part of a program that must execute atomically (i.e. entirely without interruption, or not at all)

Race Condition Example

Race condition example:

```
int iBalance = 2000;
…
static void addBonus(int iSig) {
    iBalance += 50;
}
int main(void) {
   signal(SIGINT, addBonus);
   …
   iBalance += 100;
   …
```

To save slide space, we ignore error handling here and subsequently

Race Condition Example (cont.)

Race condition example in assembly language

```
int iBalance = 2000;
…
void addBonus(int iSig) {
    iBalance += 50;
}
int main(void) {
   signal(SIGINT, addBonus);
   …
   iBalance += 100;
   …
```

```
movl iBalance, %ecx
addl $50, %ecx
movl %ecx, iBalance
```

```
movl iBalance, %eax
addl $100, %eax
movl %eax, iBalance
```

Let's say the compiler generates that assembly language code

Race Condition Example (cont.)

(1) main() begins to execute

```
int iBalance = 2000;
…
void addBonus(int iSig) {
    iBalance += 50;
}
int main(void) {
    signal(SIGINT, addBonus);
    …
    iBalance += 100;
    …
```

```
movl iBalance, %ecx
addl $50, %ecx
movl %ecx, iBalance
```

```
movl iBalance, %eax
addl $100, %eax
movl %eax, iBalance
```

2000

2100

Race Condition Example (cont.)

(2) SIGINT signal arrives; control transfers to addBonus()

```
int iBalance = 2000;
…
void addBonus(int iSig) {
    iBalance += 50;
}
int main(void) {
    signal(SIGINT, addBonus);
    …
    iBalance += 100;
    …
```

```
movl iBalance, %ecx
addl $50, %ecx
movl %ecx, iBalance
```

2000
2050
2050

```
movl iBalance, %eax
addl $100, %eax
movl %eax, iBalance
```

2000
2100

Race Condition Example (cont.)

(3) addBonus() terminates; control returns to main()

```
int iBalance = 2000;
…
void addBonus(int iSig) {
    iBalance += 50;
}
int main(void) {
    signal(SIGINT, addBonus);
    …
    iBalance += 100;
    …
```

```
movl iBalance, %ecx
addl $50, %ecx
movl %ecx, iBalance
```

2000
2050
2050

```
movl iBalance, %eax
addl $100, %eax
movl %eax, iBalance
```

2000
2100
2100

Lost $50 !!!

Critical Sections

Solution: Must make sure that **critical sections** of code are not interrupted

```
int iBalance = 2000;
…
void addBonus(int iSig) {
    iBalance += 50;
}
int main(void) {
    signal(SIGINT, addBonus);
    …
    iBalance += 100;
    …
```

Critical section

Critical section

Outline

1. Unix Process Control
2. Signals
3. Sending Signals
4. Handling Signals
5. Race Conditions and Critical Sections
6. **Blocking Signals**
7. Alarms
8. (If time) Interval Timers
9. Conclusion

Blocking Signals

Blocking signals

- **Blocking** a signal **queues** it for delivery at a later time
- Differs from **ignoring** a signal

Each process has a **signal mask** in the kernel

- OS uses the mask to decide which signals to deliver
- User program can modify mask with **`sigprocmask()`**

Function for Blocking Signals

sigprocmask()

```
int sigprocmask(int iHow,
                const sigset_t *psSet,
                sigset_t *psOldSet);
```

- **psSet**: Pointer to a signal set
- **psOldSet**: (Irrelevant for our purposes)
- **iHow**: How to modify the signal mask
  - **SIG_BLOCK**: Add **psSet** to the current mask
  - **SIG_UNBLOCK**: Remove **psSet** from the current mask
  - **SIG_SETMASK**: Install **psSet** as the signal mask
- Returns 0 iff successful

Functions for constructing signal sets

- **sigemptyset(), sigaddset(),** …

Blocking Signals Example


```
int main(void) {
    sigset_t sSet;
    signal(SIGINT, addBonus);
    …
    sigemptyset(&sSet);
    sigaddset(&sSet, SIGINT);
    sigprocmask(SIG_BLOCK, &sSet, NULL);
    iBalance += 100;
    sigprocmask(SIG_UNBLOCK, &sSet, NULL);
    …
}
```

Block SIGINT signals

Critical section

Unblock SIGINT signals

Blocking Signals in Handlers

How to block signals when handler is executing?

- While executing a handler for a signal of type x, all signals of type x are blocked automatically
- When/if signal handler returns, block is removed

```
void addBonus(int iSig) {
    iBalance += 50;
}
```

SIGINT signals automatically blocked in SIGINT handler

Outline

1. Unix Process Control
2. Signals
3. Sending Signals
4. Handling Signals
5. Race Conditions and Critical Sections
6. Blocking Signals
7. **Alarms**
8. (If time) Interval Timers
9. Conclusion

Alarms

`alarm()`

```
unsigned int alarm(unsigned int uiSec);
```

- Sends 14/SIGALRM signal after **`uiSec`** seconds
- Cancels pending alarm if **`uiSec`** is 0
- Uses **real time**, alias **wall-clock time**
  - Time spent executing other processes counts
  - Time spent waiting for user input counts
- Return value is irrelevant for our purposes

Used to implement time-outs

Alarm Example 1

Program testalarm.c:

```
#define _GNU_SOURCE
#include <stdio.h>
#include <assert.h>
#include <signal.h>
#include <unistd.h>

static void myHandler(int iSig) {
   printf("In myHandler with argument %d\n", iSig);

   /* Set another alarm. */
   alarm(2);
}
…
```

Alarm Example 1 (cont.)

Program testalarm.c (cont.):

```
…
int main(void)
{
    sigset_t sSet;

    /* Make sure SIGALRM signals are not blocked. */
    sigemptyset(&sSet);
    sigaddset(&sSet, SIGALRM);
    sigprocmask(SIG_UNBLOCK, &sSet, NULL);

    signal(SIGALRM, myHandler);
    …
```

Safe, but shouldn't be necessary; compensates for a Linux bug

Alarm Example 1 (cont.)

Program testalarm.c (cont.):

```
    …

    /* Set an alarm. */
    alarm(2);

    printf("Entering an infinite loop\n");
    for (;;)
        ;

    return 0;
}
```

Alarm Example 1 (cont.)

[Demo of testalarm.c]

Alarm Example 2

Program testalarmtimeout.c:

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <signal.h>
#include <unistd.h>

static void myHandler(int iSig)
{
   printf("\nSorry.  You took too long.\n");
   exit(EXIT_FAILURE);
}
```

Alarm Example 2 (cont.)

Program testalarmtimeout.c (cont.):

```
int main(void) {
    int i;
    sigset_t sSet;

    /* Make sure SIGALRM signals are not blocked. */
    sigemptyset(&sSet);
    sigaddset(&sSet, SIGALRM);
    sigprocmask(SIG_UNBLOCK, &sSet, NULL);

  ...
```

Safe, but shouldn't be necessary

Alarm Example 2 (cont.)

Program testalarmtimeout.c (cont.):

```
   …
   signal(SIGALRM, myHandler);

   printf("Enter a number:  ");
   alarm(5);
   scanf("%d", &i);
   alarm(0);

   printf("You entered the number %d.\n", i);
   return 0;
}
```

Alarm Example 2 (cont.)

[Demo of testalarmtimeout.c]

Outline

1. Unix Process Control
2. Signals
3. Sending Signals
4. Handling Signals
5. Race Conditions and Critical Sections
6. Blocking Signals
7. Alarms
8. **(If time) Interval Timers**
9. Conclusion

Interval Timers

`setitimer()`

```
int setitimer(int iWhich,
              const struct itimerval *psValue,
              struct itimerval *psOldValue);
```

- Sends 27/SIGPROF signal continually
- **`psValue`** specifies timing
- **`psOldValue`** is irrelevant for our purposes
- Uses **virtual time**, alias **CPU time**
  - Time spent executing other processes does not count
  - Time spent waiting for user input does not count
- Returns 0 iff successful

Used by execution profilers

Interval Timer Example

Program testitimer.c:

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <signal.h>
#include <sys/time.h>

static void myHandler(int iSig) {
   printf("In myHandler with argument %d\n", iSig);
}
…
```

Interval Timer Example (cont.)

Program testitimer.c (cont.):

```
…
int main(void)
{
    struct itimerval sTimer;

    signal(SIGPROF, myHandler);

    …
```

Interval Timer Example (cont.)

Program testitimer.c (cont.):

```
    …
    /* Send first signal in 1 second, 0 microseconds. */
    sTimer.it_value.tv_sec = 1;
    sTimer.it_value.tv_usec = 0;

    /* Send subsequent signals in 1 second,
       0 microseconds intervals. */
    sTimer.it_interval.tv_sec = 1;
    sTimer.it_interval.tv_usec = 0;

    setitimer(ITIMER_PROF, &sTimer, NULL);

    printf("Entering an infinite loop\n");
    for (;;)
       ;
    return 0;
}
```

Interval Timer Example (cont.)

[Demo of testitimer.c]

Outline

1. Unix Process Control
2. Signals
3. Sending Signals
4. Handling Signals
5. Race Conditions and Critical Sections
6. Blocking Signals
7. Alarms
8. (If time) Interval Timers
9. **Conclusion**

Predefined Signals

List of the predefined signals:

```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT     19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR      31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1
36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4  39) SIGRTMIN+5
40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8  43) SIGRTMIN+9
44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13
52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9
56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6  59) SIGRTMAX-5
60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2  63) SIGRTMAX-1
64) SIGRTMAX
```

See Bryant & O'Hallaron book for default actions, triggering exceptions

Application program can define signals with unused values

Summary

Signals

- A **signal** is an asynchronous event
- Sending signals
  - **`raise()`** or **`kill()`** **sends** a signal
- Catching signals
  - **`signal()`** **installs** a **signal handler**
  - Most signals are **catchable**
- Beware of **race conditions**
  - **`sigprocmask()`** **blocks** signals in any **critical section** of code
  - Signals of type x automatically are blocked while handler for type x signals is running

Summary (cont.)

Alarms

- Call **`alarm()`** to deliver 14/SIGALRM signals in **real/wall-clock time**
- Alarms can be used to implement **time-outs**

Interval Timers

- Call **`setitimer()`** to deliver 27/SIGPROF signals in **virtual/CPU time**
- Interval timers are used by **execution profilers**

Summary (cont.)

For more information:

Bryant & O'Hallaron, *Computer Systems: A Programmer's Perspective*, Chapter 8

Performance Improvement Revisited

Jennifer Rexford

Goals of this Lecture

- Improve program performance by exploiting knowledge of the underlying system
  - Compiler capabilities
  - Hardware architecture
  - Program execution
- And thereby:
  - Help you to write efficient programs
  - Review material from the second half of the course

When to Optimize Performance

Improving Program Performance

- Most programs are already "fast enough"
  - No need to optimize performance at all
  - Save your time, and keep the program simple/readable
- Most parts of a program are already "fast enough"
  - Usually only a small part makes the program run slowly
  - Optimize *only* this portion of the program, as needed
- Steps to improve execution (time) efficiency
  - Do timing studies (e.g., gprof)
  - Identify hot spots
  - **Optimize that part of the program**
  - Repeat as needed

Two Main Outputs of Gprof

- Call graph profile: detailed information per function
  - Which functions called it, and how much time was consumed?
  - Which functions it calls, how many times, and for how long?
  - We won't look at this output in any detail…
- Flat profile: one line per function
  - name: name of the function
  - %time: percentage of time spent executing this function
  - cumulative seconds: [skipping, as this isn't all that useful]
  - self seconds: time spent executing this function
  - calls: number of times function was called (excluding recursive)
  - self ms/call: average time per execution (excluding descendents)
  - total ms/call: average time per execution (including descendents)

Call Graph Output

```
                                called/total       parents
index  %time    self  descendents  called+self    name           index
                                called/total       children

                                                   <spontaneous>
[1]     59.7   12.97       0.00                   internal_mcount [1]
                0.00       0.00       1/3         atexit [35]
-----------------------------------------------

                                                   <spontaneous>
[2]     40.3    0.00       8.75                   _start [2]
                0.00       8.75       1/1         main [3]
                0.00       0.00       2/3         atexit [35]
-----------------------------------------------

                0.00       8.75       1/1         _start [2]
[3]     40.3    0.00       8.75       1         main [3]
                0.00       8.32       1/1         getBestMove [4]
                0.00       0.43       1/1         clock [20]
                0.00       0.00       1/747130    GameState_expandMove [6]
                0.00       0.00       1/1         exit [33]
                0.00       0.00       1/1         Move_read [36]
                0.00       0.00       1/1         GameState_new [37]
                0.00       0.00       6/747135    GameState_getStatus [31]
                0.00       0.00       1/747130    GameState_applyDeltas [25]
                0.00       0.00       1/1         GameState_write [44]
                0.00       0.00       1/1         sscanf [54]
                0.00       0.00       1/1698871   GameState_getPlayer [30]
                0.00       0.00       1/1         fprintf [58]
                0.00       0.00       1/1         Move_write [59]
                0.00       0.00       3/3         GameState_playerToStr [63]
                0.00       0.00       1/2         strcmp [65]
                0.00       0.00       1/1         GameState_playerFromStr [68]
                0.00       0.00       1/1         Move_isValid [69]
                0.00       0.00       1/1         GameState_getSearchDepth [67]
-----------------------------------------------

                0.00       8.32       1/1         main [3]
[4]     38.3    0.00       8.32       1/6       getBestMove [4]
                0.00       8.05       6/747130    minimax [5]
                0.00       0.00       6/747130    GameState_expandMove [6]
                0.00       0.00   35/4755325      Delta_free [10]
                0.00       0.00       1/2046617   GameState_genMoves [17]
                0.00       0.00       2/945027    Move_free [23]
                0.00       0.00       6/747129    GameState_applyDeltas [25]
                0.00       0.00       6/747129    GameState_unApplyDeltas [27]
                0.00       0.00       2/1698871   GameState_getPlayer [30]
-----------------------------------------------

                                 747123            minimax [5]
                0.27       8.05       6/6         getBestMove [4]
[5]     38.3    0.27       8.05       6/747123  minimax [5]
                0.63       3.56  747123/747130    GameState_expandMove [6]
                1.07       0.89 4747297/4755325   Delta_free [10]
                0.07       0.89   2046616/2046617   GameState_genMoves [17]
                0.02       0.36     945022/945027   Move_free [23]
                0.23       0.00   747123/747130     GameState_applyDeltas [25]
                0.10       0.00   747123/747129     GameState_unApplyDeltas [27]
                0.10       0.00 1698863/1698871     GameState_getPlayer [30]
                0.09       0.00   747128/747135     GameState_getStatus [31]
                0.01       0.00     542509/542509     GameState_getValue [32]
                                    747123           minimax [5]
-----------------------------------------------

                0.00       0.00       1/747130      main [3]
                0.00       0.00       6/747130      getBestMove [4]
                0.63       3.56  747123/747130      minimax [5]
[6]     19.3    0.63       3.56     747130        GameState_expandMove [6]
                0.47       2.59 4755331/5700361     calloc [7]
                0.11       0.00 2360787/2360787     .rem [28]
-----------------------------------------------

                0.00       0.00       1/5700361     Move_read [36]
                0.00       0.00       1/5700361     GameState_new [37]
                0.09       0.89     945022/5700361     GameState_genMoves [17]
                0.47       2.59 4755331/5700361     GameState_expandMove [6]
[7]     19.1    0.56       3.48 5700361           calloc [7]
                0.32       2.09 5700361/5700362     malloc [8]
                0.64       0.00 5700361/5700361     .umul [18]
                0.43       0.00 5700361/5700361     _memset [22]
                0.10       0.00 5700361/5700363     .udiv [29]
-----------------------------------------------

                0.00       0.00       1/5700362     _findbuf [41]
                0.32       2.09 5700361/5700362     calloc [7]
[8]     11.1    0.32       2.09 5700362           malloc [8]
                0.62       0.62 5700362/5700362     _malloc_unlocked              <cycle 1> [13]
                0.23       0.20 5700362/11400732    _mutex_unlock [14]
                0.22       0.20 5700362/11400732    mutex_lock [15]
```

Complex format at the beginning… let's skip for now.

Flat Profile

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
57.1	12.97	12.97				internal_mcount [1]
4.8	14.05	1.08	5700352	0.00	0.00	_free_unlocked [12]
4.4	15.04	0.99				_mcount (693)
3.5	15.84	0.80	22801464	0.00	0.00	_return_zero [16]
2.8	16.48	0.64	5700361	0.00	0.00	.umul [18]
2.8	17.11	0.63	747130	0.00	0.01	GameState_expandMove [6]
2.5	17.67	0.56	5700361	0.00	0.00	calloc [7]
2.1	18.14	0.47	11400732	0.00	0.00	_mutex_unlock [14]
1.9	18.58	0.44	11400732	0.00	0.00	mutex_lock [15]
1.9	19.01	0.43	5700361	0.00	0.00	_memset [22]
1.9	19.44	0.43	1	430.00	430.00	.div [21]
1.8	19.85	0.41	5157853	0.00	0.00	cleanfree [19]
1.4	20.17	0.32	5700366	0.00	0.00	_malloc_unlocked [13]
1.4	20.49	0.32	5700362	0.00	0.00	malloc [8]
1.3	20.79	0.30	5157847	0.00	0.00	_smalloc [24]
1.2	21.06	0.27	6	45.00	1386.66	minimax [5]
1.1	21.31	0.25	4755325	0.00	0.00	Delta_free [10]
1.0	21.54	0.23	5700352	0.00	0.00	free [9]
1.0	21.77	0.23	747130	0.00	0.00	GameState_applyDeltas [25]
1.0	21.99	0.22	5157845	0.00	0.00	realfree [26]
1.0	22.21	0.22	747129	0.00	0.00	GameState_unApplyDeltas [27]
0.5	22.32	0.11	2360787	0.00	0.00	.rem [28]
0.4	22.42	0.10	5700363	0.00	0.00	.udiv [29]
0.4	22.52	0.10	1698871	0.00	0.00	GameState_getPlayer [30]
0.4	22.61	0.09	747135	0.00	0.00	GameState_getStatus [31]
0.3	22.68	0.07	204617	0.00	0.00	GameState_genMoves [17]
0.1	22.70	0.02	945027	0.00	0.00	Move_free [23]
0.0	22.71	0.01	542509	0.00	0.00	GameState_getValue [32]
0.0	22.71	0.00	104	0.00	0.00	_ferror_unlocked [357]
0.0	22.71	0.00	64	0.00	0.00	_realbufend [358]
0.0	22.71	0.00	54	0.00	0.00	nvmatch [60]
0.0	22.71	0.00	52	0.00	0.00	_doprnt [42]
0.0	22.71	0.00	51	0.00	0.00	memchr [61]
0.0	22.71	0.00	51	0.00	0.00	printf [43]
0.0	22.71	0.00	13	0.00	0.00	_write [359]
0.0	22.71	0.00	10	0.00	0.00	_xflsbuf [360]
0.0	22.71	0.00	7	0.00	0.00	_memcpy [361]
0.0	22.71	0.00	4	0.00	0.00	.mul [62]
0.0	22.71	0.00	4	0.00	0.00	__errno [362]
0.0	22.71	0.00	4	0.00	0.00	_fflush_u [363]
0.0	22.71	0.00	3	0.00	0.00	GameState_playerToStr [63]
0.0	22.71	0.00	3	0.00	0.00	_findbuf [41]

Second part of profile looks like this; it's the simple (i.e.,useful) part; corresponds to the "prof" tool

Overhead of Profiling

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
57.1	**12.97**	**12.97**				**internal_mcount**
4.8	14.05	1.08	5700352	0.00	0.00	_free_unlocked
4.4	**15.04**	**0.99**				**_mcount (693)**
3.5	15.84	0.80	22801464	0.00	0.00	_return_zero
2.8	16.48	0.64	5700361	0.00	0.00	.umul [18]
2.8	17.11	0.63	747130	0.00	0.01	GameState_expa
2.5	17.67	0.56	5700361	0.00	0.00	calloc [7]
2.1	18.14	0.47	11400732	0.00	0.00	_mutex_unlock
1.9	18.58	0.44	11400732	0.00	0.00	mutex_lock
1.9	19.01	0.43	5700361	0.00	0.00	_memset [22]
1.9	19.44	0.43	1	430.00	430.00	.div [21]
1.8	19.85	0.41	5157853	0.00	0.00	cleanfree [19]
1.4	20.17	0.32	5700366	0.00	0.00	_malloc_unlo
1.4	20.49	0.32	5700362	0.00	0.00	malloc [8]
1.3	20.79	0.30	5157847	0.00	0.00	_smalloc
1.2	21.06	0.27	6	45.00	1386.66	minimax [5]
1.1	21.31	0.25	4755325	0.00	0.00	Delta_free [10]
1.0	21.54	0.23	5700352	0.00	0.00	free [9]
1.0	21.77	0.23	747130	0.00	0.00	GameState_appl
1.0	21.99	0.22	5157845	0.00	0.00	realfree [26]
1.0	22.21	0.22	747129	0.00	0.00	GameState_unAp
0.5	22.32	0.11	2360787	0.00	0.00	.rem [28]
0.4	22.42	0.10	5700363	0.00	0.00	.udiv [29]
0.4	22.52	0.10	1698871	0.00	0.00	GameState_getPl
0.4	22.61	0.09	747135	0.00	0.00	GameState_getSt

Malloc/calloc/free/...

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
57.1	12.97	12.97				internal_mcount [1]
4.8	**14.05**	**1.08**	**5700352**	**0.00**	**0.00**	**_free_unlocked [12]**
4.4	15.04	0.99				_mcount (693)
3.5	15.84	0.80	22801464	0.00	0.00	_return_zero [16]
2.8	16.48	0.64	5700361	0.00	0.00	.umul [18]
2.8	17.11	0.63	747130	0.00	0.01	GameState_expandMove
2.5	**17.67**	**0.56**	**5700361**	**0.00**	**0.00**	**calloc [7]**
2.1	**18.14**	**0.47**	**11400732**	**0.00**	**0.00**	**_mutex_unlock [14]**
1.9	**18.58**	**0.44**	**11400732**	**0.00**	**0.00**	**mutex_lock [15]**
1.9	**19.01**	**0.43**	**5700361**	**0.00**	**0.00**	**_memset [22]**
1.9	19.44	0.43	1	430.00	430.00	.div [21]
1.8	**19.85**	**0.41**	**5157853**	**0.00**	**0.00**	**cleanfree [19]**
1.4	**20.17**	**0.32**	**5700366**	**0.00**	**0.00**	**_malloc_unlocked [13]**
1.4	**20.49**	**0.32**	**5700362**	**0.00**	**0.00**	**malloc [8]**
1.3	**20.79**	**0.30**	**5157847**	**0.00**	**0.00**	**_smalloc [24]**
1.2	21.06	0.27	6	45.00	1386.66	minimax [5]
1.1	21.31	0.25	4755325	0.00	0.00	Delta_free [10]
1.0	**21.54**	**0.23**	**5700352**	**0.00**	**0.00**	**free [9]**
1.0	21.77	0.23	747130	0.00	0.00	GameState_applyDeltas
1.0	**21.99**	**0.22**	**5157845**	**0.00**	**0.00**	**realfree [26]**
1.0	22.21	0.22	747129	0.00	0.00	GameState_unApplyDeltas
0.5	22.32	0.11	2360787	0.00	0.00	.rem [28]
0.4	22.42	0.10	5700363	0.00	0.00	.udiv [29]
0.4	22.52	0.10	1698871	0.00	0.00	GameState_getPlayer
0.4	22.61	0.09	747135	0.00	0.00	GameState_getStatus
0.3	22.68	0.07	204617	0.00	0.00	GameState_genMoves [17]

expandMove

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
57.1	12.97	12.97				internal_mcount [1]
4.8	14.05	1.08	5700352	0.00	0.00	_free_unlocked [12]
4.4	15.04	0.99				_mcount (693)
3.5	15.84	0.80	22801464	0.00	0.00	_return_zero [16]
2.8	16.48	0.64	5700361	0.00	0.00	.umul [18]
2.8	**17.11**	**0.63**	**747130**	**0.00**	**0.01**	**GameState_expandMove**
2.5	17.67	0.56	5700361	0.00	0.00	calloc [7]
2.1	18.14	0.47	11400732	0.00	0.00	_mutex_unlock [14]
1.9	18.58	0.44	11400732	0.00	0.00	mutex_lock [15]
1.9	19.01	0.43	5700361	0.00	0.00	_memset [22]
1.9	19.44	0.43	1	430.00	430.00	.div [21]
1.8	19.85	0.41	5157853	0.00	0.00	cleanfree [19]
1.4	20.17	0.32	5700366	0.00	0.00	_malloc_unlocked [13]
1.4	20.49	0.32	5700362	0.00	0.00	malloc [8]
1.3	20.79	0.30	5157847	0.00	0.00	_smalloc [24]
1.2	21.06	0.27	6	45.00	1386.66	minimax [5]
1.1	21.31	0.25	4755325	0.00	0.00	Delta_free [10]
1.0	21.54	0.23	5700352	0.00	0.00	free [9]
1.0	21.77	0.23	747130	0.00	0.00	GameState_applyDeltas
1.0	21.99	0.22	5157845	0.00	0.00	realfree [26]

May be worthwhile to optimize this routine

Don't Even *Think* of Optimizing These

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
57.1	12.97	12.97				internal_mcount [1]
4.8	14.05	1.08	5700352	0.00	0.00	_free_unlocked [12]
4.4	15.04	0.99				_mcount (693)
3.5	15.84	0.80	22801464	0.00	0.00	_return_zero [16]
2.8	16.48	0.64	5700361	0.00	0.00	.umul [18]
2.8	17.11	0.63	747130	0.00	0.01	GameState_expandMove [6]
2.5	17.67	0.56	5700361	0.00	0.00	calloc [7]
2.1	18.14	0.47	11400732	0.00	0.00	_mutex_unlock [14]
1.9	18.58	0.44	11400732	0.00	0.00	mutex_lock [15]
1.9	19.01	0.43	5700361	0.00	0.00	_memset [22]
1.9	19.44	0.43	1	430.00	430.00	.div [21]
1.8	19.85	0.41	5157853	0.00	0.00	cleanfree [19]
1.4	20.17	0.32	5700366	0.00	0.00	_malloc_unlocked <cycle 1> [13]
1.4	20.49	0.32	5700362	0.00	0.00	malloc [8]
1.3	20.79	0.30	5157847	0.00	0.00	_smalloc <cycle 1> [24]
1.2	21.06	0.27	6	45.00	1386.66	minimax [5]
1.1	21.31	0.25	4755325	0.00	0.00	Delta_free [10]
1.0	21.54	0.23	5700352	0.00	0.00	free [9]
1.0	21.77	0.23	747130	0.00	0.00	GameState_applyDeltas [25]
1.0	21.99	0.22	5157845	0.00	0.00	realfree [26]
1.0	22.21	0.22	747129	0.00	0.00	GameState_unApplyDeltas [27]
0.5	22.32	0.11	2360787	0.00	0.00	.rem [28]
0.4	22.42	0.10	5700363	0.00	0.00	.udiv [29]
0.4	22.52	0.10	1698871	0.00	0.00	GameState_getPlayer [30]
0.4	22.61	0.09	747135	0.00	0.00	GameState_getStatus [31]
0.3	22.68	0.07	204617	0.00	0.00	GameState_genMoves [17]
0.1	22.70	0.02	945027	0.00	0.00	Move_free [23]
0.0	22.71	0.01	542509	0.00	0.00	GameState_getValue [32]
0.0	22.71	0.00	104	0.00	0.00	_ferror_unlocked [357]
0.0	22.71	0.00	4	0.00	0.00	_thr_main [367]
0.0	22.71	0.00	3	0.00	0.00	GameState_playerToStr [63]
0.0	22.71	0.00	2	0.00	0.00	strcmp [66]
0.0	22.71	0.00	1	0.00	0.00	GameState_getSearchDepth [67]
0.0	22.71	0.00	1	0.00	0.00	GameState_new [37]
0.0	22.71	0.00	1	0.00	0.00	GameState_playerFromStr [68]
0.0	22.71	0.00	1	0.00	0.00	GameState_write [44]
0.0	22.71	0.00	1	0.00	0.00	Move_isValid [69]
0.0	22.71	0.00	1	0.00	0.00	Move_read [36]
0.0	22.71	0.00	1	0.00	0.00	Move_write [59]
0.0	22.71	0.00	1	0.00	0.00	check_nlspath_env [46]
0.0	22.71	0.00	1	0.00	430.00	clock [20]
0.0	22.71	0.00	1	0.00	0.00	exit [33]
0.0	22.71	0.00	1	0.00	8319.99	getBestMove [4]
0.0	22.71	0.00	1	0.00	0.00	getenv [47]
0.0	22.71	0.00	1	0.00	8750.00	main [3]
0.0	22.71	0.00	1	0.00	0.00	mem_init [70]
0.0	22.71	0.00	1	0.00	0.00	number [71]
0.0	22.71	0.00	1	0.00	0.00	scanf [53]

Ways to Optimize Performance

- Better data structures and algorithms
  - Improves the *"asymptotic complexity"*
    - Better scaling of computation/storage as input grows
    - E.g., going from $O(n^2)$ sorting algorithm to $O(n \log n)$
  - Clearly important if large inputs are expected
  - Requires understanding data structures and algorithms
- Better source code the compiler can optimize
  - Improves the *"constant factors"*
    - Faster computation during each iteration of a loop
    - E.g., going from *1000n* to *10n* running time
  - Clearly important if a portion of code is running slowly
  - Requires understanding hardware, compiler, execution

Helping the Compiler Do Its Job

Optimizing Compilers

- Provide efficient mapping of program to machine
  - Register allocation
  - Code selection and ordering
  - Eliminating minor inefficiencies
- Don't (usually) improve asymptotic efficiency
  - Up to the programmer to select best overall algorithm
- Have difficulty overcoming "optimization blockers"
  - Potential function side-effects
  - Potential memory aliasing

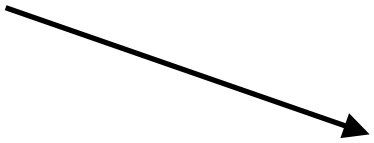
Limitations of Optimizing Compilers

- Fundamental constraint
  - Compiler must not change program behavior
  - Ever, even under rare pathological inputs
- Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles
  - Data ranges more limited than variable types suggest
  - Array elements remain unchanged by function calls
- Most analysis is performed only within functions
  - Whole-program analysis is too expensive in most cases
- Most analysis is based only on static information
  - Compiler has difficulty anticipating run-time inputs

Avoiding Repeated Computation

- A good compiler recognizes simple optimizations
  - Avoiding redundant computations in simple loops
  - Still, programmer may still want to make it explicit
- Example
  - Repetition of computation: n * i

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```

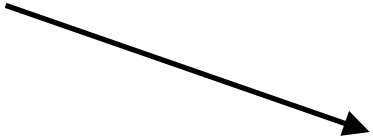


```
for (i = 0; i < n; i++) {
  ni = n * i;
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
}
```

Worrying About Side Effects

- Compiler cannot always avoid repeated computation
  - May not know if the code has a "side effect"
  - … that makes the transformation change the code's behavior
- Is this transformation okay?

```
int func1(int x) {
  return f(x) + f(x) + f(x) + f(x);
}
```



```
int func1(int x) {
  return 4 * f(x);
}
```

- Not necessarily, if

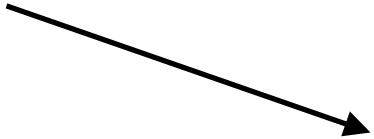
```
int counter = 0;

int f(int x) {
  return counter++;
}
```

And this function may be defined in another file known only at link time!

Another Example on Side Effects

- Is this optimization okay?

```
for (i = 0; i < strlen(s); i++) {
   /* Do something with s[i] */
}
```



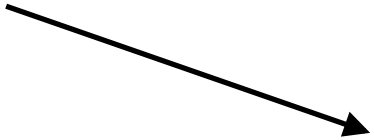
```
length = strlen(s);
for (i = 0; i < length; i++) {
   /* Do something with s[i] */
}
```

- Short answer: it depends
  - Compiler often cannot tell
  - Most compilers do not try to identify side effects
- Programmer knows best
  - And can decide whether the optimization is safe

Memory Aliasing

- Is this optimization okay?

```
void twiddle(int *xp, int *yp) {
    *xp += *yp;
    *xp += *yp;
}
```

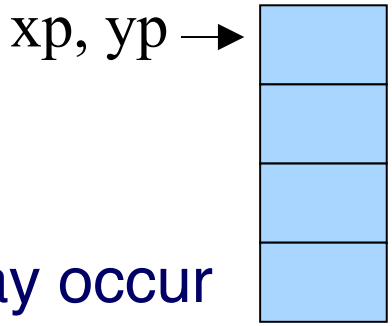


```
void twiddle(int *xp, int *yp) {
    *xp += 2 * *yp;
}
```

- Not necessarily, what if xp and yp are equal?
  - First version: result is 4 times *xp
  - Second version: result is 3 times *xp

Memory Aliasing

- Memory aliasing
  - Single data location accessed through multiple names
  - E.g., two pointers that point to the same memory location
- Modifying the data using one name
  - Implicitly modifies the values seen through other names

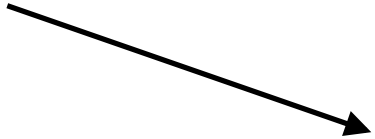


- Blocks optimization by the compiler
  - The compiler cannot tell when aliasing may occur
  - … and so must forgo optimizing the code
- Programmer often *does* know
  - And *can* optimize the code accordingly

Another Aliasing Example

- Is this optimization okay?

```
int *x, *y;
…
*x = 5;
*y = 10;
printf("x=%d\n", *x);
```



```
printf("x=5\n");
```

- Not necessarily
  - If y and x point to the same location in memory…
  - … the correct output is "x = 10\n"

Summary: Helping the Compiler

- Compiler can perform many optimizations
  - Register allocation
  - Code selection and ordering
  - Eliminating minor inefficiencies
- But often the compiler needs your help
  - Knowing if code is free of side effects
  - Knowing if memory aliasing will not happen
- Modifying the code can lead to better performance
  - Profile the code to identify the “hot spots”
  - Look at the assembly language the compiler produces
  - Rewrite the code to get the compiler to do the right thing

Exploiting the Hardware

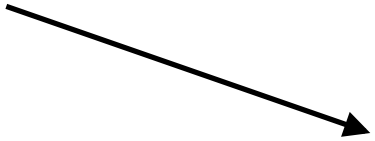
Underlying Hardware

- Implements a collection of instructions
  - Instruction set varies from one architecture to another
  - Some instructions may be faster than others
- Registers and caches are faster than main memory
  - Number of registers and sizes of caches vary
  - Exploiting both spatial and temporal locality
- Exploits opportunities for parallelism
  - Pipelining: decoding one instruction while running another
    - Benefits from code that runs in a sequence
  - Superscalar: perform multiple operations per clock cycle
    - Benefits from operations that can run independently
  - Speculative execution: performing instructions before knowing they will be reached (e.g., without knowing outcome of a branch)

Addition Faster Than Multiplication

- Adding instead of multiplying
  - Addition is faster than multiplication
- Recognize sequences of products
  - Replace multiplication with repeated addition

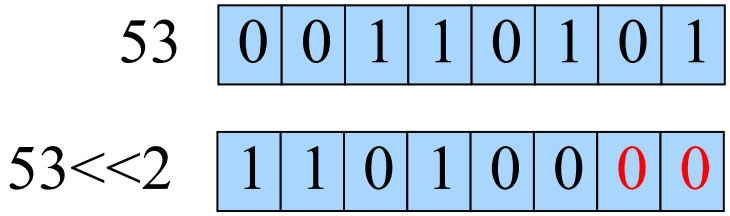
```
for (i = 0; i < n; i++) {
  ni = n * i;
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
}
```



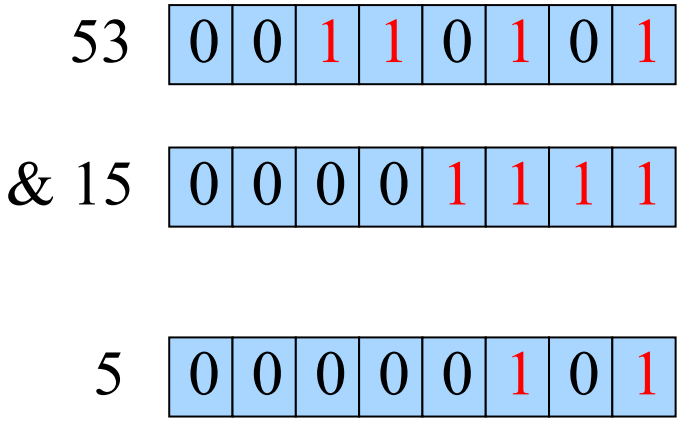
```
ni = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
  ni += n;
}
```

Bit Operations Faster Than Arithmetic

- Shift operations to multiple/divide by powers of 2
  - "x >> 3" is faster than "x/8"
  - "x << 3" is faster than "x * 8"



- Bit masking is faster than mod operation
  - "x & 15" is faster than "x % 16"



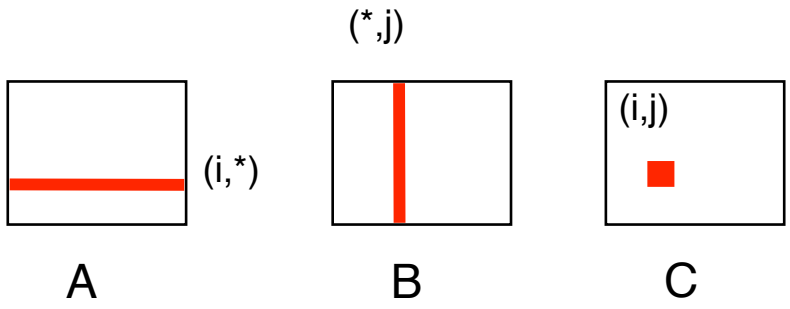
Caching: Matrix Multiplication

- Caches
  - Slower than registers, but faster than main memory
  - Both instruction caches and data caches
- Locality
  - Temporal locality: recently-referenced items are likely to be referenced in near future
  - Spatial locality: Items with nearby addresses tend to be referenced close together in time
- Matrix multiplication
  - Multiply n-by-n matrices A and B, and store in matrix C
  - Performance heavily depends on effective use of caches

Matrix Multiply: Cache Effects

```
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    for (k=0; k<n; k++)
      c[i][j] += a[i][k] * b[k][j];
  }
}
```

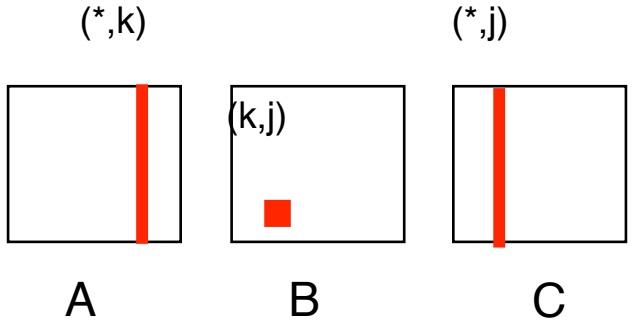
- Reasonable cache effects
  - Good spatial locality for A
  - Poor spatial locality for B
  - Good temporal locality for C



Matrix Multiply: Cache Effects

```
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * b[k][j];
  }
}
```

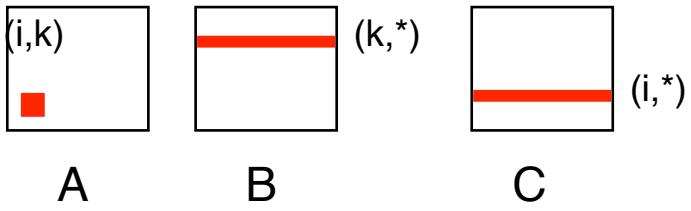
- Rather poor cache effects
  - Bad spatial locality for A
  - Good temporal locality for B
  - Bad spatial locality for C



Matrix Multiply: Cache Effects

```
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    for (j=0; j<n; j++)
      c[i][j] += a[i][k] * b[k][j];
  }
}
```

- Good cache effects
  - Good temporal locality for A
  - Good spatial locality for B
  - Good spatial locality for C



Parallelism: Loop Unrolling

- What limits the performance?

```
for (i = 0; i < length; i++)
  sum += data[i];
```

- Limited apparent parallelism
  - One main operation per iteration (plus book-keeping)
  - Not enough work to keep multiple functional units busy
  - Disruption of instruction pipeline from frequent branches

- Solution: unroll the loop
  - Perform multiple operations on each iteration

Parallelism: After Loop Unrolling

- Original code

```
for (i = 0; i < length; i++)
  sum += data[i];
```

- After loop unrolling (by three)

```
/* Combine three elements at a time */
limit = length - 2;
for (i = 0; i < limit; i+=3)
  sum += data[i] + data[i+1] + data[i+2];

/* Finish any remaining elements */
for ( ; i < length; i++)
  sum += data[i];
```

Program Execution

Avoiding Function Calls

- Function calls are expensive
  - Caller saves registers and pushes arguments on stack
  - Callee saves registers and pushes local variables on stack
  - Call and return disrupt the sequence flow of the code
- Function inlining:

```
void g(void) {
   /* Some code */
}

void f(void) {
   …
   g();
   …
}
```

Some compilers support "inline" keyword directive.

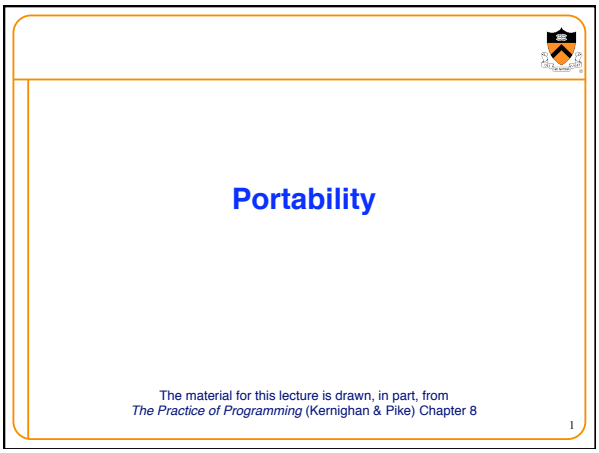
```
void f(void) {
   …
   /* Some code */
   …
}
```

Writing Your Own Malloc and Free

- Dynamic memory management
  - `malloc()` to allocate blocks of memory
  - `free()` to free blocks of memory
- Existing `malloc()` and `free()` implementations
  - Designed to handle a wide range of request sizes
  - Good most of the time, but rarely the best for all workloads
- Designing your own dynamic memory management
  - Forego using traditional `malloc()` and `free()`, and write your own
  - E.g., if you know all blocks will be the same size
  - E.g., if you know blocks will usually be freed in the order allocated
  - E.g., <insert your known special property here>

Conclusion

- Work smarter, not harder
  - No need to optimize a program that is “fast enough”
  - Optimize only when, and where, necessary
- Speeding up a program
  - Better data structures and algorithms: better asymptotic behavior (the “COS 226 way”)
  - Optimized code: smaller constants (the “COS 217 way”)
- Techniques for speeding up a program
  - Coax the compiler
  - Exploit capabilities of the hardware
  - Capitalize on knowledge of program execution



Portability

The material for this lecture is drawn, in part, from
The Practice of Programming (Kernighan & Pike) Chapter 8

1

Goals of this Lecture

- Learn to write code that works with multiple:
  - Hardware platforms
  - Operating systems
  - Compilers
  - Human cultures
- Why?
  - Moving existing code to a new context is easier/cheaper than writing new code for the new context
  - Code that is portable is (by definition) easier to move; portability reduces software costs
  - Relative to other high-level languages (e.g., Java), C is notoriously non-portable

2

The Real World is Heterogeneous

- Multiple kinds of hardware
  - 32-bit Intel Architecture
  - 64-bit IA, PowerPC, Sparc, MIPS, Arms, …
- Multiple operating systems
  - Linux
  - Windows, Mac, Sun, AIX, …
- Multiple character sets
  - ASCII
  - Latin-1, Unicode, …
- Multiple human alphabets and languages

3

Portability

- Goal: Run program on any system
  - No modifications to source code required
  - Program continues to perform correctly
    - Ideally, the program performs well too

4

C is Notoriously Non-Portable

- Recall C design goals…
  - Create Unix operating system and associated software
  - Reasonably "high level", but…
  - Close to the hardware for efficiency
- So C90 is underspecified
  - Compiler designer has freedom to reflect the design of the underlying hardware
- But hardware systems differ!
  - So C compilers differ
- Extra care is required to write portable C code

5

Structure of This Talk

- General heuristics
- Heuristics for handling differences
  - Hardware
  - OS
  - Compiler
  - Library
  - Cultural
- General themes
  - Be aware of your assumptions
  - Avoid being too clever

6

General Heuristics

Some general portability heuristics…

7

Intersection

(1) Program to the intersection

- Use only features that are common to all target environments
- I.e., program to the ***intersection*** of features, not the ***union***

- When that's not possible…

8

Encapsulation

(2) Encapsulate

- Localize and encapsulate features that are not in the intersection
- Use parallel ***source code*** files -- so non-intersection code can be chosen at ***link-time***
- Use parallel ***data*** files – so non-intersection data (e.g. textual messages) can be chosen at ***run-time***

- When that's not possible, as a last resort…

9

Conditional Compilation

(3) Use conditional compilation

```
#ifdef __UNIX__
  /* Unix-specific code */
#endif
…
#ifdef __WINDOWS__
   /* MS Windows-specific code */
#endif
…
```

- And above all…

10

Test!!!

(4) Test the program with multiple:
- Hardware (Intel, MIPS, SPARC, …)
- Operating systems (Linux, Solaris, MS Windows, …)
- Compilers (GNU, MS Visual Studio, …)
- Cultures (United States, Europe, Asia, …)

11

Hardware Differences

- Some **hardware** differences, and corresponding portability heuristics…

12

Natural Word Size

- Obstacle: Natural word size
  - In some systems, natural word size is 4 bytes
  - In some (esp. older) systems, natural word size is 2 bytes
  - In some (esp. newer) systems, natural word size is 8 bytes
- C90 intentionally does not specify sizeof(int); depends upon natural word size of underlying hardware

13

Natural Word Size (cont.)

(5) Don't assume data type sizes

- Not portable:

```
int *p;
…
p = malloc(4);
…
```

- Portable:

```
int *p;
…
p = malloc(sizeof(int));
…
```

14

Right Shift

- Obstacle: Right shift operation
  - In some systems, right shift operation is **logical**
    - Right shift of a negative signed int fills with **zeroes**
  - In some systems, right shift operation is **arithmetic**
    - Right shift of a negative signed int fills with **ones**
- C90 intentionally does not specify semantics of right shift; depends upon right shift operator of underlying hardware

15

Right Shift (cont.)

(6) Don't right-shift signed ints

- Not portable:

```
…
-3 >> 1
…
```

Logical shift => 2147483646
Arithmetic shift => -2

- Portable:

```
…
/* Don't do that!!! */
…
```

- But if you must: shift and mask/set high bit as needed

16

Byte Order

- Obstacle: Byte order
  - Some systems (e.g. Intel) use little endian byte order
    - Least significant byte of a multi-byte entity is stored at lowest memory address

The int 5 at address 1000:

1000	00000101
1001	00000000
1002	00000000
1003	00000000

  - Some systems (e.g. SPARC) use big endian byte order
    - Most significant byte of a multi-byte entity is stored at lowest memory address

The int 5 at address 1000:

1000	00000000
1001	00000000
1002	00000000
1003	00000101

17

Byte Order (cont.)

(7) Don't rely on byte order in code

- Not portable:

```
int i = 5;
char c;
…
c = *(char*)&i; /* Silly, but legal */
```

Little endian:
c = 5
Big endian:
c = 0;

- Portable:

```
int i = 5;
char c;
…
/* Don't do that! Or... */
c = (char)i;
```

- But if you must extract bytes: shift and mask

18

Byte Order (cont.)

(8) Use text for data exchange

- Not portable:

`fwrite()` writes raw data to a file

Run on a ***little*** endian computer

```
unsigned short s = 5;
FILE *f = fopen("myfile", "w");
fwrite(&s, sizeof(unsigned short), 1, f);
```

myfile: `00000101 00000000`

`fread()` reads raw data from a file

Run on a ***big*** endian computer: Reads 1280!!!

```
unsigned short s;
FILE *f = fopen("myfile", "r");
fread(&s, sizeof(unsigned short), 1, f);
```

19

Byte Order (cont.)

- Portable:

`fprintf()` converts raw data to ASCII text

Run on a ***big or little*** endian computer

```
unsigned short s = 5;
FILE *f = fopen("myfile", "w");
fprintf(f, "%hu", s);
```

myfile: `00110101` ASCII code for '5'

`fscanf()` reads ASCII text and converts to raw data

Run on a ***big or little*** endian computer: Reads 5

```
unsigned short s;
FILE *f = fopen("myfile", "r");
fscanf(f, "%hu", &s);
```

20

Byte Order (cont.)

If you must exchange raw data…

(9) Write and read one byte at a time

Run on a ***big or little*** endian computer

```
unsigned short s = 5;
FILE *f = fopen("myfile", "w");
fputc(s >> 8, f);   /* high-order byte */
fputc(s & 0xFF, f); /* low-order byte */
```

Decide on big-endian data exchange format

myfile: `00000000 00000101`

Run on a ***big or little*** endian computer: Reads 5

```
unsigned short s;
FILE *f = fopen("myfile", "r");
s = fgetc(f) << 8;    /* high-order byte */
s |= fgetc(f) & 0xFF; /* low-order byte */
```

21

OS Differences

- Some **operating system** differences, and corresponding portability heuristics…

22

End-of-Line Characters

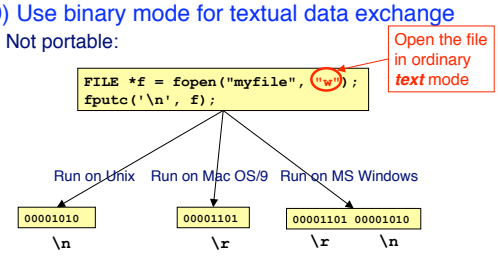
- Obstacle: Representation of "end-of-line"
  - **Unix** (including Mac OS/X) represents end-of-line as 1 byte: **00001010** (binary)
  - **Mac OS/9** represents end-of-line as 1 byte: **00001101** (binary)
  - **MS Windows** represents end-of-line as 2 bytes: **00001101 00001010** (binary)

23

End-of-Line Characters (cont.)

(10) Use binary mode for textual data exchange

- Not portable:



- Trouble if read via fgetc() on "wrong" operating system

24

End-of-Line Characters (cont.)

- Portable:

```
FILE *f = fopen("myfile", "wb");
fputc('\n', f);
```

Open the file in ***binary*** mode

Run on Unix, Mac OS/9, or MS Windows

```
00001010
  \n
```

- No problem if read via fgetc() in binary mode on "wrong" operating system
- I.e., there is no "wrong" operating system!

Data Alignment

- Obstacle: Data alignment
  - Some hardware requires data to be aligned on particular boundaries
  - Some operating systems impose additional constraints:

OS	char	short	int	double
Linux	1	2	4	4
MS Windows	1	2	4	8

Start address must be evenly divisible by:

  - Moreover…
  - If a structure must begin on an x-byte boundary, then it also must end on an x-byte boundary
    - Implication: Some structures must contain padding

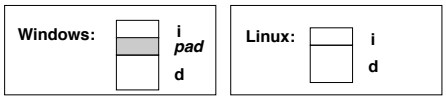
Data Alignment (cont.)

(11) Don't rely on data alignment

- Not portable:

Allocates 12 bytes; too few bytes on MS Windows

```
struct S {
   int i;
   double d;
}
…
struct S *p;
…
p = (struct S*)malloc(sizeof(int)+sizeof(double));
```

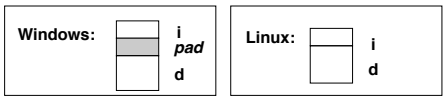


Data Alignment (cont.)

- Portable:

```
struct S {
   int i;
   double d;
}
…
struct S *p;
…
p = (struct S*) malloc(sizeof(struct S));
```

Allocates
- 12 bytes on Linux
- 16 bytes on MS Windows



28

Character Codes

- Obstacle: Character codes
  - Some operating systems (e.g. IBM OS/390) use the EBCDIC character code
  - Some systems (e.g. Unix, MS Windows) use the ASCII character code

29

Character Codes (cont.)

(12) Don't assume ASCII

- Not portable:

```
if ((c >= 65) && (c <= 90))…
```

Assumes ASCII

- A little better:

```
if ((c >= 'A') && (c <= 'Z'))…
```

Assumes that uppercase char codes are contiguous; not true in EBCDIC

- Portable:

```
#include <ctype.h>
…
if (isupper(c))…
```

For ASCII:
(c >= 'A') && (c <= 'Z')
For EBCDIC:
((c >= 'A') && (c <= 'I'))
II ((c >= 'J') && (c <= 'R'))
II ((c >= 'S') && (c <= 'Z'))

30

Compiler Differences

- Compilers may differ because they:
  - Implement underspecified features of the C90 standard in different ways, or
  - Extend the C90 standard
- Some **compiler** differences, and corresponding portability heuristics…

31

Compiler Extensions

- Obstacle: Non-standard extensions
  - Some compilers offer non-standard extensions

32

Compiler Extensions

(13) Stick to the standard language

- For now, stick to C90 (not C99)
- Not portable:

```
…
for (int i = 0; i < 10; i++)
   …
```

Many systems allow definition of loop control variable within `for` statement, but a C90 compiler reports error

- Portable:

```
int i;
…
for (i = 0; i < 10; i++)
   …
```

33

Evaluation Order

- Obstacle: Evaluation order
  - C90 specifies that side effects and function calls must be completed at "**;**"
  - But multiple side effects within the same expression can have unpredictable results

Evaluation Order (cont.)

(14) Don't assume order of evaluation

- Not portable:

```
strings[i] = names[++i];
```

`i` is incremented before indexing `names`; but has `i` been incremented before indexing `strings`? C90 doesn't say

- Portable (either of these, as intended):

```
i++;
strings[i] = names[i];
```

```
strings[i] = names[i+1];
i++;
```


Evaluation Order (cont.)

- Not portable:

Which call of `getchar()` is executed first? C90 doesn't say

```
printf("%c %c\n", getchar(), getchar());
```

- Portable (either of these, as intended):

```
i = getchar();
j = getchar();
printf("%c %c\n", i, j);
```

```
i = getchar();
j = getchar();
printf("%c %c\n", j, i);
```

Char Signedness

- Obstacle: Char signedness
  - C90 does not specify signedness of char
  - On some systems, char means signed char
  - On other systems, char means unsigned char

37

Char Signedness (cont.)

(15) Don't assume signedness of char

- If necessary, specify "signed char" or "unsigned char"
- Not portable:

```
int a[256];
char c;
c = (char)255;
…
… a[c] …
```

If char is unsigned, then a[c] is a[255]
=> fine
If char is signed, then a[c] is a[-1]
=> out of bounds

- Portable:

```
int a[256];
unsigned char c;
c = 255;
…
… a[c] …
```

38

Char Signedness (cont.)

- Not portable:

```
int i;
char s[MAX+1];
for (i = 0; i < MAX; i++)
   if ((s[i] = getchar()) == '\n') || (s[i] == EOF))
      break;
s[i] = '\0';
```

If char is unsigned, then this always is FALSE

- Portable:

```
int c, i;
char s[MAX+1];
for (i = 0; i < MAX; i++) {
   if ((c = getchar()) == '\n') || (c == EOF))
      break;
   s[i] = c;
}
s[i] = '\0';
```

39

Library Differences

- Some **library** differences, and corresponding portability heuristics…

40

Library Extensions

- Obstacle: Non-standard functions
  - "Standard" libraries bundled with some development environments (e.g. GNU, MS Visual Studio) offer non-standard functions

41

Library Extensions

(16) Stick to the standard library functions

- For now, stick to the C90 standard library functions
- Not portable:

```
char *s = "hello";
char *copy;
…
copy = strdup(s);
…
```

`strdup()` is available in many "standard" libraries, but is not defined in C90

- Portable:

```
char *s = "hello";
char *copy;
…
copy = (char*)malloc(strlen(s) + 1);
strcpy(copy, s);
```

- Alternative: write your own strdup if needed

42

Cultural Differences

- Some **cultural** differences, and corresponding portability heuristics…

43

Character Code Size

- Obstacle: Character code size
  - United States
    - Alphabet requires 7 bits => 1 byte per character
    - Popular character code: ASCII
  - Western Europe
    - Alphabets require 8 bits => 1 byte per character
    - Popular character code: Latin-1
  - China, Japan, Korea, etc.
    - Alphabets require 16 bits => 2 bytes per character
    - Popular character code: Unicode

44

Character Code Size

(17) Don't assume 1-byte character code size

- Not portable:

```
char c = 'a';
```

- Portable:
  - C90 has ***no good solution***
  - C99 has "wide character" data type, constants, and associated functions

```
#include <stddef.h>
…
wchar_t c = L'\x3B1'; /* Greek lower case alpha */
```

  - But then beware of byte-order portability problems!
  - Future is not promising

45

Human Language

- Obstacle: Humans speak different natural languages!

46

Human Language (cont.)

(18) Don't assume English

- Not portable:

```
/* somefile.c */

…
printf("Bad input");
…
```

- Can't avoid natural language! So…

47

Human Language (cont.)

- Encapsulate code

```
/* somefile.c */

#include "messages.h"
…
printf(getMsg(5));
…
```

- Choose appropriate "message.c" file at ***link-time***

Messages module, with multiple implementations

```
/* messages.h */
char *getMsg(int msgNum);
```

```
/* englishmessages.c */
char *getMsg(int msgNum) {
   switch(msgNum) {
      …
      case 5:
         return "Bad input";
      …
   }
}
```

```
/* spanishmessages.c */
char *getMsg(int msgNum) {
   switch(msgNum) {
      …
      case 5:
         return "Mala entrada";
      …
   }
}
```

48

Human Language (cont.)

- Maybe even better: encapsulate ***data***

Messages module

```
/* messages.h */
char *getMsg(int msgNum);
```

```
/* messages.c */

enum {MSG_COUNT = 100};
char *getMsg(int msgNum) {
   static char *msg[MSG_COUNT];
   static int firstCall = 1;
   if (firstCall) {
      <Read all messages from
       appropriate messages.txt
       file into msg>
      firstCall = 0;
   }
   return msg[msgNum];
}
```

```
/* englishmessages.txt */

…
Bad input
…
```

```
/* spanishmessages.txt */

…
Mala entrada
…
```

- Choose appropriate "message.txt" file at ***run-time***

Summary

- General heuristics
  - (1) Program to the intersection
  - (2) Encapsulate
  - (3) Use conditional compilation (as a last resort)
  - (4) Test!!!

Summary (cont.)

- Heuristics related to **hardware** differences
  - (5) Don't assume data type sizes
  - (6) Don't right-shift signed ints
  - (7) Don't rely on byte order in code
  - (8) Use text for data exchange
  - (9) Write and read 1 byte at a time
- Heuristics related to **OS** differences
  - (10) Use binary mode for textual data exchange
  - (11) Don't rely on data alignment
  - (12) Don't assume ASCII

Summary (cont.)

- Heuristics related to **compiler** differences
  - (13) Stick to the standard language
  - (14) Don't assume evaluation order
  - (15) Don't assume signedness of char
- Heuristic related to **library** differences
  - (16) Stick to the standard library
- Heuristics related to **cultural** differences
  - (17) Don't assume 1-byte char code size
  - (18) Don't assume English

52